MASTER IN COMPUTER SCIENCE

Abstract Interpretation of Prolog programs
Optimizations of an implementation

Englebert, Vincent; Roland, Didier

Award date:
1992

Awarding institution:
University of Namur

Facultés Universitaires Notre-Dame de la Paix, Namur
Institut d'Informatique
Année académique 1991-1992

Abstract Interpretation of Prolog programs: Optimizations of an implementation

Vincent ENGLEBERT
Didier ROLAND

Mémoire présenté en vue de l'obtention du grade
de Licencié et Maître en Informatique

Abstract Interpretation of Prolog programs

Vincent ENGLEBERT—Didier ROLAND

Résumé du mémoire

L'interprétation abstraite de programmes Prolog est une technique d'analyse statique de ces programmes réalisée dans le but de pouvoir améliorer les compilateurs Prolog. Mais cette analyse est lourde, elle demande beaucoup de temps et de mémoire. Partant d'un premier programme d'interprétation abstraite requérant au moins une station de travail, nous avons implémenté plusieurs optimisations dont deux retiennent particulièrement notre attention. La première est basée sur une étude plus approfondie du déroulement de l'algorithme. La seconde est plus générale et vise à cacher les opérations répétitives les plus longues. Ces optimisations permettent d'avoir un gain de temps et de mémoire considérable, au point de pouvoir exécuter le programme sur un ordinateur personnel. Enfin, deux autres techniques, l'une basée sur la détection de structures strictement croissantes, l'autre sur la réexécution de certains buts, permettent également d'obtenir des résultats plus précis. Ce mémoire développe toutes ces optimisations et discute les résultats obtenus sur deux domains différents, avec toute une batterie de programmes tests.

Abstract

Abstract interpretation of Prolog programs is a technique for static analysis of these programs that is achieved in order to improve Prolog compilers. But this technique is arduous, it consumes a lot of time and a lot of memory. On the basis of an original abstract interpretation program that requires at least a workstation, we have implemented several optimizations among which two especially retain our attention. The first one is based on a thorough examination of the execution of the algorithm. The second one is more general, it aims at caching the longest repetitive operations. These optimizations allow a considerable gain of execution time and of memory space and make possible to run the program on a personal computer. Finally, two other techniques, one based on the detection of strictly increasing structures, the other based on the reexecution of some goals, also allow to obtain more accurate results. This report develop all these optimizations and discuss the results on two different domains, with a series of test programs.

Contents

List of Figures

List of Tables

Preface

Abstract interpretation is a promising technique for static analysis of computer programs, and especially in logic programming. More and more people are studying this technique and a few implementations are appearing. The team of researchers of B. Le Charlier is working on it for a few years. A lot of work have already been done. Some implementations have already been realized. One of them have retain our attention. It works well, its results are incouraging, but it needs fast machines with a big amount of memory. Our purpose, in this report, is to optimize that version, to make it able to be run faster on smaller computers.

Firstly, we present the theorical background underlying abstract interpretation. A few words of introduction are based on [1]. Then an introduction to the fixpoint theory is based on [18]. And finally, the introduction to normalized Prolog programs, abstract interpretation and to the fixpoint based abstract interpretation of Prolog programs has been written on the basis of [13].

In chapter 3, we present two domains of values for objects handled by the abstract interpretation algorithm. These domains are extracted of [20].

Then comes a description of the original program. All the definitions of functions and all the algorithms are those presented in [20, 13]. The original implementation is presented in [15]. The first part of the optimization was a translation of that Pascal original version in C language. With that translation, a few optimizations have already been done by both V. Englebert and D. Roland. Between all these optimizations, one is described more in detail by D. Roland because of the importance of the data structure involved.

Then we get to the heart of the matter. The first genuine optimization, with clause prefixes, has been realized by D. Roland. And, the second one, caching of the most time consumming operations , has been achieved by V. Englebert. The comparison of all the results obtained by these optimizations has been written by both of them in common with P. Van Hentenryck. Two other techniques for an increased accuracy of the results have been implemented too. Detection of increasing structures has been done by D. Roland and the reexecution of some sub-goals is the work of V. Englebert.

The following articles are the main ones we used as the basis of the work:

- The PhD thesis of K. Musumbu ([20]) is the theoretical basis for most of other papers. It contains a detailed description of all the abstract functions

used for abstract interpretation of Prolog programs, a detailed description of abstract domains and a detailed description of the basic algorithms for abstract interpretation of Prolog programs, as well as all the proofs of consistency of the domains and the functions and correctness of the algorithms.

- "Efficient and Accurate Algorithms for the Abstract Interpretation of Prolog Programs" ([13]), written by B. Le Charlier, K. Musumbu and P. Van Hentenryck, contains, firstly, a very good and complete summary of the PhD thesis of K. Musumbu and, secondly, new optimized algorithms with all their complexity analysis.
- "Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog" ([15]), written by B. Le Charlier and P. Van Hentenryck contains a complete description of the original program and a lot of results about its efficiency and its accurateness.

People interested by the programs can contact Pascal Van Hentenryck by E-mail at `pvh@cs.brown.edu`

Acknowledgments

First of all, we would like to thank Pascal Van Hentenryck who invited us in Brown University and who encouraged us in the work and Baudouin Le Charlier who teached us a large amounts of things during five years and who made us share the pleasure of programming with elegance. Our stay in Providence has been full-filled with a lot of good moments we had with Pascal Van Hentenryck and his wife, with Susan Griswold Treter and Esther Rogers. We would like to thank all these people too. And, last but not least, our parents and all other people who supported us during all that time.

Chapter 1

Abstract interpretation

This chapter is a summary of the theoretical background about fixpoint theory and abstract interpretation. All the ideas and definitions are extracted of [18, 13, 20].

1.1 Introduction

In the recent years, programmers have developped a number of techniques to analyse their algorithms in order to improve them without changing the semantics. Among these methods, one will require our attention in this paper: Abstract interpretation. It is a static method which studies a program to infer some interesting properties from its source without executing it.

The way of doing this consists of assigning to variables and arguments properties on their possible values and, from them, to deduce properties on the result that would be obtained with an execution. If we define a concrete domain as the set of all values an object can take in a given language, we can define an abstract domain as a subset of the set of parts of the concrete domain. This subset is the reflect of the properties we are interested in. For instance, all programming languages have an integer type. The concrete domain corresponding to that type is a range of integers. An abstract domain could be the following: $\{0, +, -, \emptyset, any\}$ where $+$ stands for all strictly positive integers, 0 for the integer 0, $-$ for all strictly negative integers, *any* for any integer and $\emptyset$ for no value at all.

In order to deduce properties on the result of a program, we need to redefine the operations of the language for these abstract domains. For instance, let's assume that our language is able to perform simple arithmetics on integers. Wich one can be redefined this way:

+	$\emptyset$	0	+	−	*any*
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
0	$\emptyset$	0	+	−	*any*
+	$\emptyset$	+	+	*any*	*any*
−	$\emptyset$	−	*any*	−	*any*
any	$\emptyset$	*any*	*any*	*any*	*any*

/	$\emptyset$	0	+	−	*any*
$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
0	$\emptyset$	$\emptyset$	0	0	*any*
+	$\emptyset$	$\emptyset$	+	−	*any*
−	$\emptyset$	$\emptyset$	−	+	*any*
any	$\emptyset$	*any*	*any*	*any*	*any*

Thus if we have defined an abstract domain and redefined all the operations on this one, then it seems sufficient to reexecute the program in this new context to obtain the expected results. Unfortunaly no theoric frame allows to prove that the program is correct or even that it ends.

The *fixpoint* theory gives a frame to the abstract interpretation and a tool as well to compute the result. The fixpoint was first used to exprime recursive function in lambda calculus but can also be useful in a lot of others domains like abstact interpretation.

1.2 Fixpoint theory

The semantic of a Prolog program P can be defined as the least fixed point of a transformation associated to P. First of all let us define the fixpoint theory principles.

Let E be a set endowed with a partial order relation $\leq$.

a. a *chain* in $(E, \leq)$ is an increasing sequence $(x_i)_{i=0}^{\infty}$ verifying $x_i \leq x_{i+1}\ \forall i \geq 0$.

b. $(E, \leq)$ is a *cpo* (complete partial order) if there exists a least element noted $\perp$ and if all chains have a least upper bound noted $\sqcup_{n=0}^{\infty}\ x_n$.

c. Let $f : (E, \leq) \longrightarrow (F, \leq)$ a function. f is *monotonic* iff

$$\forall x, y \in E \quad x \leq y \Longrightarrow f(x) \leq f(y)$$

d. Let $f : (E, \leq) \longrightarrow (F, \leq)$ a function. f is *continuous* iff

- f is monotonic
- for every chain $(e_i)_0^{\infty}$, we have

$$f(\sqcup_{i=0}^{\infty} e_i) = \bigsqcup_{i=0}^{\infty} f(e_i)$$

or, equivalently:

- for all chain $(e_i)_0^\infty$, we have

$$\bigsqcup_{i=0}^{\infty} f(e_i) \; exists \Longrightarrow f(\sqcup_{i=0}^{\infty} e_i) = \bigsqcup_{i=0}^{\infty} f(e_i)$$

e. Let $f : E \longrightarrow E$ be a function. f has a fixed point iff $\exists y : y = f(y)$

We can now present the Fixpoint Theorem:

> Let $(X, \leq)$ be a cpo. If the function $f : X \longrightarrow X$ is continuous then f has a least fixpoint noted $\mu(f)$.
>
> **Note:** $\mu(f)$ may be computed as $\bigsqcup_{i=0}^{\infty} f^n(\bot)$

However the continuity property is required to insure the existence of the fixpoint, this condition is not very hard. In english, the continuity expresses that a function (in computer science context) defined on a functional domain invokes its argument a finite number of times each time it has to compute a result.

$$f : (Z^+ \longrightarrow Z^+) \longrightarrow X \text{ is continuous}$$

iff

$$\forall g \in (Z^+ \longrightarrow Z^+) \exists D finite : f(g) = f(g_D)$$

where g_D is the function g restricted to the subdomain D.

How can the fixpoint theory be useful? Let $\mathcal{L}$ be a language and $P \in \mathcal{L}$ a program. Although P is a syntaxic object, its execution can generate an information.

> For example: the information/semantic associated to a functional program is a table of pairs (argument,result).

Let $\overline{\mathcal{L}}$ be the set of all the informations produced by any program of $\mathcal{L}$ and $\overline{P}$ the semantic of the program P.

> For example: the set of tables with all possible values for the pairs (argument,result).

Now let us suppose that it exists a transformation associated to the program P: τ_P defined as:

$$\tau_P : \overline{\mathcal{L}} \longrightarrow \overline{\mathcal{L}}$$

which has the following property:

$$\tau_P(\overline{P}) = \overline{P} \text{where } \overline{P} \text{ is the least fixpoint}$$

then if $\overline{\mathcal{L}}$ is a cpo and if τ_P is continuous then $\overline{P}$ can be systematically computed as

$$\bigsqcup_{n=0}^{\infty} {\tau_P}^n(\bot)$$

The following sections will show how to build a such transformation automaticaly from Prolog programs and how to build its abstract counterpart. Unfortunately, it is impossible to compute systematically the exact least fixpoint for every program as it is proved in program theory[1] We may however approximate the least fixpoint.

1.3 Prolog

The fixpoint theory summarized in the previous section can be applied to logic programming and Prolog. The algorithm described later handles Prolog programs or, more precisely, normalized Prolog programs. It is not a difficult matter to translate any Prolog program into its normalized version. The advantage of the normalized form comes from the fact that a substitution for a goal p/n is always expressed in terms of variables $X_1, \ldots, X_n$. This greatly simplifies all the traditionnal problems encountered with renaming.

Normalized Prolog programs are built from an ordered set of program variables

$$\{X_1, X_2, \ldots, X_i, \ldots\}$$

and is composed of clauses of the form

$$H : -B_1, \ldots, B_p$$

where H is the head and $B_1, \ldots, B_p$ is the body. If a clause contains m variables, these are necessarily $X_1, \ldots, X_m$. Moreover, the head of the clause is an atom $p(X_1, \ldots, X_n)$ where p is a predicate symbol of arity n, noted p/n. The subgoals in the body of the clause are of the form:

- $q(X_{i_1}, \ldots, X_{i_m})$ where $i_1, \ldots, i_m$ are distinct indices
- $X_{i_1} = X_{i_2}$ with $i_1 \neq i_2$
- $X_{i_1} = f(X_{i_2}, \ldots, X_{i_m})$ where f is a function of arity $m-1$ and $i_1, \ldots, i_m$ are distinct indices

The first form is called a procedure call. The second and third forms, called built-ins, enable to achieve unification. Additional built-in predicates (e.g. arithmetic primitives) can be accommodated in the framework but are not discussed here for simplicity.

Figure 1.1 and figure 1.2 respectively describe, as an example, a Prolog program and its normalized translation. This program, `APPEND/3`, will be used throughout this paper.

[1] If the exact semantic can be computed then it is possible to determine if a program ends or not, among other things. But this property is undecidable!

```
append([],L,L).
append([H|T1],L,[H|T2]):-append(T1,L,T2).
```

Figure 1.1: Prolog code for APPEND

```
append(X1,X2,X3) :-
   X1 = [],
   X3 = X2.
append(X1,X2,X3) :-
   X1 = [X4 | X5],
   X3 = [X4 | X6],
   append(X5,X2,X6).
```

Figure 1.2: Normalized version of APPEND

1.4 Abstract Interpretation of Prolog

It is now time to define abstract interpretation more precisely, with regard to Prolog. First, it is necessary to define the concrete and asbtract domains and their respective operations. Next, we will present the transformation we already spoke about.

Definition 1 Let PV be an infinite set of program variables. Let D be a subset of PV. Then $CS_D = \{\Theta : \forall\theta \in \Theta, dom(\theta) = D$ and Θ is complete$\}$ is a cpo with respect to set inclusion $(CS_D, \subseteq)$, where Θ is complete means $\forall\theta \in \Theta$, θ and θ' are equivalent ($\exists\sigma, \sigma'$ such that $\theta = \theta'\sigma$ and $\sigma' = \theta\sigma$) implies that $\theta' \in \Theta$. $\Theta \in CS_D$ are sets of Prolog program variables substitutions.

Definition 2 $\{AS_D : D \subset PV \wedge (AS_D, \leq)$ is a cpo$\}$ is the abstract domain. $\beta \in AS_D$ are abstract substitutions.

Definition 3 Let P be a normalized Prolog program. Let's denote *sat* a set of abstract tuples, i.e. a set of tuples of the form $(\beta_{in}, p, \beta_{out})$ where $\beta_{in}, \beta_{out} \in AS_D$, D being the set of program variables appearing in the predicate p. SAT will be the set of all functionnal *sat*, i.e. all *sat*s for which

$$\forall(\beta, p), \exists \text{ at most one } \beta', \text{ noted } sat(\beta, p), \text{ such that } (\beta, p, \beta') \in sat$$

A correspondance between the abstract domain and the concrete domain can now be established through a concretization function.

Definition 4 A *concretization function* for an abstract domain AS_D is a monotonic and continuous function $Cc : AS_D \longrightarrow CS_D$ where

- monotonic means: $\forall \beta_1, \beta_2, \beta_1 \subseteq \beta_2, \forall p$ for which $sat(\beta_1, p)$ and $sat(\beta_2, p)$ are defined, $sat(\beta_1, p) \subseteq sat(\beta_2, p)$
- *sat* is total iff $sat(\beta, p)$ is defined for all (β, p), $p \in P$ and $\beta \in AS_D$, D being the set of program variables of p
- continuous means: if *sat* is total, monotonic and if any chain $\beta_1, \beta_2, \ldots, \beta_n$ satisfies:

$$sat(\bigcup_{i=1}^{\infty}\{\beta_i\}, p) = \bigcup_{i=1}^{\infty}\{sat(\beta_i, p)\}$$

And lastly, abstract operations can be defined:

Definition 5 An abstract operation is a function $ao : AS_{D_1} \times \ldots \times AS_{D_n} \longrightarrow AS_D$ consistent with respect to a concrete operation $o : CS_{D_1} \times \ldots \times CS_{D_n} \longrightarrow CS_D$ if and only if

$\forall \beta_1 \in AS_{D_1}, \ldots, \beta_n \in AS_{D_n} : o(Cc(\beta_1), \ldots, Cc(\beta_n)) \subseteq Cc(ao(\beta_1, \ldots, \beta_n))$.

An abstract semantics of the language can now be established by assigning to each function an abstract counterpart.

In Prolog, elements of CS_D are called *concrete substitutions* and elements of AS_D *abstract substitutions*. Figure 1.3 shows the abstract transformation. This transformation uses a number of abstract operations:

- `UNION`$\{\beta_1, \ldots, \beta_n\}$ where $\beta_1, \ldots, \beta_n$ are abstract substitutions from the same cpo: this operation returns an abstract substitution representing all the substitutions satisfying at least one β_i. It is used to compute the output of a procedure given the outputs for its clauses.
- `AI_VAR(`β`)` where β is an abstract substitution on $\{X_1, X_2\}$: this operation returns the abstract substitution obtained from β by unifying variables X_1, X_2. It is used for goals of the form $X_i = X_j$ in normalized programs.
- `AI_FUNC(`β, f`)` where β is an abstract substitution on $\{X_1, \ldots, X_n\}$ and f is a function symbol of arity $n-1$: this operation returns the abstract substitution obtained from β by unifying X_1 and $f(X_2, \ldots, X_n)$. It is used for goals $X_{i_1} = f(X_{i_2}, \ldots, X_{i_n})$ in normalized programs.
- `EXTC(`c, β`)` where β is an abstract substitution on $\{X_1, \ldots, X_n\}$ and c is a clause containing variables $\{X_1, \ldots, X_m\}$ $(m \geq n)$: this operation returns the abstract substitution obtained by extending β to accommodate the new free variables of the clause. It is used at the entry of a clause to include the variables in the body not present in the head. In logical terms, this operation, together with the next operation, achieves the role of the existential quantifier.

- RESTRC(c, β) where β is an abstract substitution on the clause variables $\{X_1, \ldots, X_m\}$ and $\{X_1, \ldots, X_n\}$ are the head variables of clause c ($n \leq m$): this operation returns the abstract substitution obtained by projecting β on variables $\{X_1, \ldots, X_n\}$. It is used at the exit of a clause to restrict the substitution to the head variables only.

- RESTRG(g, β) where β is an abstract substitution on $D = \{X_1, \ldots, X_n\}$, and g is a goal $p(X_{i_1}, \ldots, X_{i_m})$ (or $X_{i_1} = X_{i_2}$ or $X_{i_1} = f(X_{i_2}, \ldots, X_{i_m})$): this operation returns the abstract substitution obtained by

  1. projecting β on $\{X_{i_1}, \ldots, X_{i_m}\}$ obtaining β';
  2. expressing β' in terms of $\{X_1, \ldots, X_m\}$ by mapping X_{i_k} to X_k.

  It is used before the execution of a goal in the body of a clause. The resulting substitution is expressed in terms of $\{X_1, \ldots, X_m\}$, i.e. in the same way as the input and output substitutions of p in the abstract domain.

- EXTG(g, β, β') where β is an abstract substitution on $D = \{X_1, \ldots, X_n\}$, the variables of the clause where g appears, g is a goal $p(X_{i_1}, \ldots, X_{i_m})$ (or $X_{i_1} = X_{i_2}$ or $X_{i_1} = f(X_{i_2}, \ldots, X_{i_m})$) with $\{X_{i_1}, \ldots, X_{i_m}\} \subseteq D$ and β' is an abstract substitution on $\{X_1, \ldots, X_m\}$ representing the result of $p(X_1, \ldots, X_n)$ β'' where β''=RESTRG(g, β): this operation returns the abstract substitution obtained by extending β to take into account the result β' of the goal g. It is used after the execution of a goal to propagate the results of the goal on the substitution for all variables of the clause.

- EXTEND(β, p, sat), given an abstract substitution β, a predicate symbol p, and a set of abstract tuples sat which does not contain (β, p) in its domain, returns a set of abstract tuples sat' containing (β, p) in its domain. Moreover the value $sat'(\beta, p)$ is defined as the lub (i.e. the least upper bound) of all $sat(\beta', p)$ such that $\beta' \leq \beta$.

- ADJUST(β, p, β', sat), where β' represents a new result computed for the pair (β, p), returns a sat' which is sat updated with this new result. More precisely, the value of $sat'(\beta'', p)$ for all $\beta'' \geq \beta$ is equal to $lub\{\beta', sat(\beta'', p)\}$ and all other values are left unchanged. In the algorithm, we use a slightly more general version of ADJUST which, in addition to the new set of abstract tuples, returns the set of pairs (β, p) the values of which have been updated.

All these definitions are extracted from [13]. They are the minimum required to understand the following. For more complete and more general definitions, the reader can refer to that article.

Note: We have spoken about "substitutions" and "abstract substitutions" as two different things, the second ones being the abstraction of the first ones. In the following of this report, when no confusion is possible, the term "substitution"

will sometimes be used in place of "abstract substitution" for the simplicity of writing when used often. Anyway, when greek letters are used, θ will always denote substitutions, while α or β will always denote abstract substitutions.

1.5 Fixpoint Based Abstract Interpretation

A number of elements $sat \in SAT$ are monotonic, continuous and total. Let $SCAT$ be the set of those elements.

An abstract transformation $TSAT : SCAT \longrightarrow SCAT$ can then be defined consistent with respect to the program P, using abstract functions defined on AS_D and $SCAT$ (figure 1.3), assuming that,

- $UD = \{(\beta, p) : \beta \in AS_D, D \text{ being the set of arguments of } p \in P\}$,
- $\bot = \{(\beta, p, \emptyset) : (\beta, p) \in UD\}$
- $sat_1 \sqsubseteq sat_2 \Longleftrightarrow \forall (\beta, p) \in UD,\ sat_1(\beta, p) \sqsubseteq sat_2(\beta, p)$

Proposition 6 $(SCAT, \sqsubseteq)$ is a cpo.
Proof Let $(sat_i)_{i=0}^{\infty}$ be a chain in $SCAT$. Then

$$\forall (\beta, p) \in UD,\ sat_i(\beta, p) \sqsubseteq sat_{i+1}(\beta, p) \forall i \geq 0$$

Let us fix β and p and let us define

$$\gamma_i = sat_i(\beta, p)\ \forall i \geq 0$$

$(\gamma_i)_{i=0}^{\infty}$ is a chain in SA_D and since SA_D is a cpo, $\bigsqcup_{i=0}^{\infty} \gamma_i$ exists and let us note $\gamma = \bigsqcup_{i=0}^{\infty} \gamma_i$. We build a new sat defined as

$$\forall (\beta, p) \in UD : sat(\beta, p) = \bigsqcup_{i=0}^{\infty} sat_i(\beta, p)$$

It is now clear that $SCAT$ is a cpo.
□

The abstract interpretation of the program P is the search for the least fixpoint of the associated abstract transformation $TSAT$: $\mu(TSAT)$, or, more precisely, of a set of elements of $\mu(TSAT)$ that are relevant for the evaluation of a particular query of P: (β_{in}, p).

$$TSAT(sat) = \{(\beta, p, \beta') : (\beta, p) \in AD \text{ and } \beta' = T_p(\beta, p, sat)\}$$

$T_p(\beta, p, sat) = UNION(\beta_1, \ldots, \beta_n)$
where $\beta_i = T_c(\beta, c_i, sat)$,
$c_1, \ldots, c_n$ are clauses of p.

$T_c(\beta, c, sat) = RESTRC(c, \beta')$
where $\beta' = T_b(EXTC(c, \beta), b, sat)$,
b is the body of c.

$T_b(\beta, <>, sat) = \beta$
$T_b(\beta, g.gs, sat) = T_b(\beta_3, gs, sat)$
where $\beta_3 = EXTG(g, \beta, \beta_2)$,

$$\beta_2 = \begin{cases} sat(\beta_1, p) & \text{if } g \text{ is } p(\ldots) \\ AI\_VAR(\beta_1) & \text{if } g \text{ is } x_i = x_j \\ AI\_FUNC(\beta_1, f) & \text{if } g \text{ is } x_i = f(\ldots) \end{cases}$$

$\beta_1 = RESTRG(g, \beta)$

Figure 1.3: Abstract transformation

Chapter 2

Overview of the different algorithms

2.1 Domains

In the first chapter, we spoke about abstract interpretation. We said it is defined on an abstract domain, elements of which are abstract substitutions, and we defined abstract operations on this abstract domain. But, we did not define the domain very precisely, that is, we did not say what the abstract substitutions look like. We will define two different abstract domains in the next chapter. A first primitive one will be defined in order just to study the modes of the terms of the substitutions (for instance, assuming "$X \mid t$" is a binding of an abstract substitution, the mode of the term t may be *Ground*, *Var* or *Any*). And a second, more complicated domain will permit to describe the abstract substitutions more precisely. The terms of an abstract substitution are described with their type (in fact a generalized form of the mode with *Ground*, *Var*, *NoVarNorGround*, *NoGround*, *NoVar*, *GroundOrVar* and *Any*) and their form (their functor, if we know it, their arity and all their subterms).

With these domains, we will have everything we need to realize an abstract interpretation algorithm.

2.2 Algorithms

A first abstract interpretation algorithm will be described. Implemented in Pascal, translated in C with a few improvements, it will be the starting point of different optimizations. The first one is an optimization based on the structure of a Prolog program. Its goal is to avoid evaluation of clauses and of clause prefixes for which we are sure the result is the same than the last time. The second one is more independant of the problem. The same idea could be used in a variety of programs. It just consists in caching the most time consuming functions. When an operation

is performed, its result is stored. If the same operation must be done a second time or even more, its result is gotten back from the cache.

These two optimizations will be compared on the basis of their time consumption, their memory consumption and the number of operations avoided.

The chapter "Widening" will then describe a source of lack of precision and a way to avoid it. This lack of precision is due to a technique that permits to avoid infinite loops. This technique can be refined to lose precision less often.

And finally, a last optimization for a better precision. It is based on the fact that a clause is semantically equivalent if one of its subgoals is considered once or several times but the abstract interpretation algorithm is more precise if some subgoals are considered several times.

2.3 Motivations for Abstract Interpretation

To develop compilers for the Prolog language is a challenge if we expect performances as good as for compilers designed for imperative languages. Abstract interpretation can help these tools to generate a better code as it is shown in this section. Thus, it is important that abstract compilers can run rapidly with a moderate memory consumption.

The article [5] explains why the *type* and *mode* domains are useful to gain CPU time when running Prolog program and to perform garbage collection. We present here a brief overview of this article.

We suppose that the reader has a good knowledge of the Warren Abstract Machine. A complete description of WAM can be found in [2].

The Mode Domain

The *mode* domain can be useful in two ways:

- To enhance clause indexing: the general strategy used by WAM based Prolog compiler when they have to solve a goal is to try each clause associated to this goal until unification is possible, then the goal is substituted by the body of the clause inside the resolvent.

  > Here follows the sequence of WAM instructions normally produced by a compiler: try_me_else % try to unify the first clause else try another
  > retry_me_else
  > ...
  > trust_me_else fail % try to unify this clause else the attempt fails

  In some cases, the compiler can generate code which reaches directly the good clause without trying systematically these ones. It is the case for example in

the well-known program: **append**. For **append** it is sufficient that the generated code tests a *tag*[1] to decide between both clauses, the only one which can possibly be unified correctly. So a heavy unsuccessful unification attempt is avoided.

```
append([], A, A).
append([A | As], B, [C | Cs]) :-append(As, B, Cs).
```

The *list-tag*'s value ([] or [|]) is sufficient to switch between both clauses when the compiler knows that each call to **append** is made with (*ground*,*ground*,*var*).

- When unification is required, the compiler proceeds by a pattern matching algorithm, it has to know if it proceeds in *read-mode* or *write-mode* context[2]. If the *query term* is known as being fully instantiated because abstract interpretation has computed its mode as *ground* then the *write mode* will never occur. The compiler may then produce a speliazed code for operations `get_structure` and `unify_variable`. Concretely we avoid one affectation in `get_structure` and a `switch` redirection in `unify_variable`.

[1] We use here the tag defined in the WAM (see [2, pp 39–44]). Briefly, a *tag* is an additional information stocked with objects to indicate whether it is itself an object or a pointer to a more complicated structure; if a list object is empty ([]) or a *cons*-list ([. | .]) ; ...

[2] While matching, the program has to check two expressions called *query term* and *program term*. These terms ($f(X, g(a, Y), \ldots)$ for example) have a flattened representation in memory. These ones are composed of functors and of (un)instancied variables. The work begins in *read-mode*. A priori the *query term* can not get some additional information from the *program term*. The problem is to determine if both expressions match or not. Sometimes at a certain point, the procedure encounters a variable which can be specialized with a functor from the *program term*, the procedure turns on *write mode* to build the new term.

Example: unification of

query term: f(X) **program term:** f(g(X))

generates the code:

$$\left.\begin{array}{l} get\_structure\ \ f/1, X_1 \\ unify\_variable\ \ X_2 \end{array}\right\} \text{read mode}$$
$$\left.\begin{array}{l} get\_structure\ \ g/1, X_2 \\ unify\_variable\ \ X_3 \end{array}\right\} \text{write mode}$$

with respect to the expression represented on the heap as:

1: STR 2
2: f/1
3: REF 3

The Type Domain

The *type* domain provides the same advantages as the *mode* domain. But in addition an abstract substitution can indicate if a ground argument is composed or not and what the main functor is. When two clauses are characterized by an argument for which the type is different in the input substitution, it is sufficient to test the tag to switch between the two clauses.

Garbage Collection

An abstract interpretation using the *type* domain may help a compiler to do garbage collection at the compile time by computing for each variable, the last point in a clause which has updated it. So, when the compiler has to create a new variable, it can use an old variable if this variable isn't updated by the next points in the current clause.

Chapter 3

Domains

We present here both domains used to evaluate the abstract interpreter. Each section is composed of an unformal presentation followed by a formal presentation (see [20]) and finally some details of the implementation.

3.1 The *Type* domain

This domain was first defined in [20, pp II/65][1]. We present here the essential characteristics of this domain. Let us begin with an unformal presentation.

3.1.1 Unformal Presentation

We introduce here a brief overview of the abstract domain *type*. The abstract domain contains patterns (i.e. for each subterm, the main functor and a reference to its arguments is stored), sharing, same-value, and mode components.

The key concept in the representation of the substitutions in this domain is the notion of subterm. Given a substitution on a set of variables, an abstract substitution will associate the following information with each subterm:

- its *mode* (e.g. *ground*, *var*, *ngv* (i.e. neither ground nor variable));
- its *pattern* which specifies the main functor as well as the subterms which are its arguments;
- its possible *sharing* with other subterms.

Note that the *pattern* is optional. If it is omitted, the pattern is said to be undefined. In addition to the above information, each variable in the domain of the substitution is associated to one of the subterms. Note that this information enables to express that two arguments have the same value (and hence that two variables are bound together). To identify the subterms in an unambiguous way, an index is associated

[1] This one contains also the proofs of monotonicity and consistency

to each of them. If there are n subterms, we make use of indices $1, \ldots, n$. For instance, the substitution

$$\{X_1 \leftarrow t * v \ , \ X_2 \leftarrow v \ , \ X_3 \leftarrow Y_1 \setminus [\,]\}$$

will have 7 subterms. The association of indices to them could be for instance

$$\{(1, t * v), (2, t), (3, v), (4, v), (5, Y_1 \setminus [\,]), (6, Y_1), (7, [\,])\}.$$

As mentioned previously, each index is associated with a mode taken from

$$\{\bot \ , \ ground \ , \ var \ , \ ngv \ , \ novar \ , \ gv \ , \ noground \ , \ any\}.$$

In the above example, we have the following associations

$$\{(1, ground), \ (2, ground), \ (3, ground), \ (4, ground), \ (5, ngv), \ (6, var), \ (7, ground)\}.$$

The pattern component (possibly) assigns to an index an expression $f(i_1, \ldots, i_n)$ where f is a function symbol of arity n and $i_1, \ldots, i_n$ are indices. In our example, the pattern component will make the following associations

$$\{(1, 2 * 3), (2, t), (3, v), (4, v), (5, 6 \setminus 7), (7, [\,])\}.$$

Finally the sharing component specifies which indices, not associated with a pattern, may possibly share variables. We only restrict our attention to indices with no pattern since the other patterns already express some sharing information and we do not want to introduce inconsistencies between the components. The actual sharing relation can be derived from these two components. In our particular example, the only sharing is the couple $(6, 6)$ which expresses that variable Y_1 shares a variable with itself.

As the above representation may be difficult to visualize, we make use of a more appealing representation in the following. For instance, a predicate $factorize(X_1, X_2, X_3)$ instantiated by the above substitution will be represented as

$$\begin{aligned} factorize(&ground(1) : \ *(\ ground(2) : t \ , \ ground(3) : v) \ , \\ &ground(4) : v, \\ &ngv(5) : \ \setminus(\ var(6) \ , \ ground(7) : [\,]\ )) \end{aligned}$$

together with the sharing information $\{(6, 6)\}$. In the above representation, each argument is associated with a mode, with an index (between parenthesis), and with an abstract subterm after the colon. The subterm uses the same representation.

3.1.2 Formal Presentation

Modes

The diagram (see figure 3.1) describes the modes used and the order is represented by the transitive closure of this graph ($a \leq b \Leftrightarrow b \rightarrow a$). Let Ty be the set of all the modes $Ty = \{any, novar, gv, noground, ground, ngv, var, \bot\}$.
Let T be the set of all the Prolog terms and $P()$ is the set of all subsets of its arguments. The concretization function is defined as:

$$\begin{array}{ll} Cc: Ty \longrightarrow P(T) & \\ \quad ground & \{ \text{ basics terms } \} \\ \quad var & \{ \text{ variables } \} \\ \quad ngv & \text{the rest} \end{array}$$

and the last ones can be obtained from the first ones:

$$Cc(LUB(t_1, t_2)) = Cc(t_1) \cup Cc(t_2)$$
$$Cc(GLB(t_1, t_2)) = Cc(t_1) \cap Cc(t_2)$$

Same Value

The *same value* component of an abstract substitution β defined on D is a partition of D noted $sv(\beta)$. It expresses a relation between variables defined as:

$$\forall X_i, X_j \in D : sv(X_i, X_j) \Leftrightarrow \exists A \in sv(\beta) : X_i, X_j \in A$$

The concretization of the same value component is defined as:

$$\begin{array}{ll} Cc: P(D) \longrightarrow CS_D & \\ \quad sv & \{\{X_i \leftarrow t_i\}_i \mid \forall k, l \;\; sv(X_k, X_l) \Rightarrow t_k = t_l\} \end{array}$$

There is a partial order relation between same value components: let $sv_1, sv_2 \in P(D)$ then

$$sv_1 \leq sv_2 \;\Leftrightarrow\; \forall i, j : sv_2(X_i, X_j) \Rightarrow sv_1(X_i, X_j)$$

Musumbu prefers express it by a function[2] from the domain D to a finite subset of N for commodity.

The Type Component

The *type* component is a function $t_p : \{1 \ldots p\} \longrightarrow Ty - \{\bot\}$. Let us define

$$Tp_p = \{t_p \mid t_p : \{1 \ldots p\} \longrightarrow Ty - \{\bot\}\}$$

$$Tp = \bigcup_{p=0}^{\infty} Tp_p$$

The concretization is defined as:

[2]Musumbu defines $sv : N \longrightarrow N$. The link between both notations can be defined as:

$$sv(X_i, X_j) \Leftrightarrow sv(i) = sv(j)$$

$$\begin{array}{ll} Cc : Tp_p \longrightarrow & T^p \\ \quad t_p & \{(t_1, \ldots, t_p) \mid t_i \in Cc(t_p(i)) \ \ \forall i \in \{1, \ldots, p\}\} \end{array}$$

The set Tp is endowed with a partial order relation defined as:

$$\begin{array}{l} t_p : \{1, \ldots, p\} \longrightarrow Ty - \{\bot\} \\ u_{p'} : \{1, \ldots, p'\} \longrightarrow Ty - \{\bot\} \\ u_{p'} \leq t_p \Longleftrightarrow \exists f : \{1, \ldots, p\} \rightarrow \{1, \ldots, p'\} \text{ such that } u_{p'}(f(i)) \leq t_p(i) \end{array}$$

Let us note that t_p is defined on $\{1, \ldots, p\}$ where p is a priori independant from the domain of the abstract substitution. Indeed each abstract term is flattened and some components may be shared by other terms. Next sections will clear all that.

The Pattern Component

A *pattern* is a functor followed by its arguments: $i_1, \ldots, i_q$ which are positive integers. Let F_p be the set of all the patterns where $1 \leq i_1, \ldots, i_q \leq p$. The pattern component can now be defined as a partial function

$$frm : \{1, \ldots, p\} \not\longrightarrow F_p$$

where

$$\forall i \in \{1, \ldots, p\} \ \ frm(i) = f(i_1, \ldots, i_q) \Longrightarrow i \leq i_1, \ldots, i_q$$

We have spoken about optional components, for these cases:

$$frm(i) = undef$$

Let Frm_p be the set of all the pattern components defined on $\{1, \ldots, p\}$.
The concretization function is defined as:

$$\begin{array}{ll} Cc : Frm_p \longrightarrow & T^p \\ \quad frm & \{(t_1, \ldots, t_p) : \forall i \in \{1, \ldots, p\} \\ & frm(i) = f(i_1, \ldots, i_q) \Rightarrow t_i = f(t_{i_1}, \ldots, t_{i_q})\} \end{array}$$

where t_{i_j} is the i_jth component of $Cc(frm)$.

We have also a partial order relation defined as:

$$\begin{array}{l} frm : \{1, \ldots, p\} \not\longrightarrow F_p \\ frm' : \{1, \ldots, p'\} \not\longrightarrow F_{p'} \\ frm' \leq frm \Longleftrightarrow \exists g : \{1, \ldots, p\} \rightarrow \{1, \ldots, p'\} \text{ such that } frm(i) = \\ f(i_1, \ldots, i_q) \Rightarrow frm'(g(i)) = f(g(i_1), \ldots, g(i_q)) \end{array}$$

The Possible Sharing Component

The *possible sharing*, as in the *mode* domain, indicates if two abstract terms may share some variables. Thus the possible sharing (ps) is a symetric relation defined on $\{1,\ldots,p\}\times\{1,\ldots,p\}$. Let us note PS_p the set of all these relations and $PS = \bigcup_{p=0}^{\infty} PS_p$.
If frm is a pattern defined on $\{1,\ldots,p\}$ then ps is compatible with frm iff

$$ps(i,j) \Longrightarrow frm(i) = frm(j) = undef \quad \forall i,j \in \{1,\ldots,p\}$$

ps was until now defined only for undefined subterms. ps can be extended to all other subterms by computing its transitive closure: $ps*$

- $ps(i,j) \Longrightarrow ps*(i,j)$
- $frm(k) = f(\ldots,j,\ldots)$ and $ps*(i,j) \Longrightarrow ps*(k,i)\ \forall i,j,k \in \{1,\ldots,p\}$

Let us note PS_{frm} the set of all possible ps components compatible with the pattern frm.
The concretization function is defined as:

$$\begin{array}{lll} Cc_1 : PS_p & \longrightarrow & T^p \\ \quad ps & & \{(t_1,\ldots,t_p) : \forall i,j \in \{1,\ldots,p\} \\ & & var(t_i) \cap var(t_j) \neq \emptyset \Rightarrow ps(i,j)\} \end{array}$$

$$\begin{array}{lll} Cc_2 : PS_{frm} & \longrightarrow & T^p \\ \quad ps & & \{(t_1,\ldots,t_p) : \forall i,j \in \{1,\ldots,p\} \text{ and } frm(i) = frm(j) = undef \\ & & \text{we have } var(t_i) \cap var(t_j) \neq \emptyset \Rightarrow ps(i,j)\} \end{array}$$

PS_p and PS_{frm} are endowed with a partial order relation defined as:

If $ps \in PS_p$ and $ps' \in PS_{p'}$
$ps' \leq ps \Longleftrightarrow$

$\exists g : \{1,\ldots,p\} \longrightarrow \{1,\ldots,p'\}$ such that
$\forall i,j \in \{1,\ldots,p\} : ps'(g(i),g(j)) \Rightarrow ps(i,j)$

If frm and frm' are patterns defined on $\{1,\ldots,p\}$ and $\{1,\ldots,p'\}$. If $ps \in PS_{frm}$ and $ps' \in PS_{frm'}$ $ps' \leq ps \Longleftrightarrow$

$\forall i,j \in \{1,\ldots,p\}$ and $frm(i) = frm(j) = undef$
$ps' * (g(i),g(j)) \Rightarrow ps(i,j)$

Abstract Substitutions

The abstract domain AS_D is a subset of $SV \times Tp \times Frm \times PS$. And the following constraint must be satisfied:

$(sv, tp, frm, ps) \in AS_D \iff$

1. $\exists m, p \in N : m \leq p$ and $sv \in SV_m$ and $tp \in Tp_p$ and $ps \in PS_{frm}$
2. $\forall i \in \{1, \ldots, p\} : frm(i) = f(i_1, \ldots, i_n) \Longrightarrow$
   - $tp(i) \leq cons(f, tp(i_1), \ldots, tp(i_n))$
   - $tp(i_1), \ldots, tp(i_n) \leq extr(f, tp(i))$
3. $\forall i : m \leq i \leq p : \exists j \in \{1, \ldots, p\} : frm(j) = f(\ldots, i, \ldots)$

Operations *cons* and *extr* are defined as:

$cons(f, T_1, \ldots, T_n) = T' \iff$
$T_1, \ldots, T_n \in Ty$
$t_1, \ldots, t_n$ are all terms
f/n is a functor
$\forall i \in \{1, \ldots, n\} \;\; t_i \in Cc(T_i) \Rightarrow f(t_1, \ldots, t_n) \in Cc(T')$

$extr(T, f) = (T_1, \ldots, T_n) \iff$
$T, T_1, \ldots, T_n \in Ty$ and f/n is a functor
$t_1, \ldots, t_n$ are all terms
$f(t_1, \ldots, t_n) \in Cc(T) \Rightarrow (\forall i \in \{1, \ldots, n\} \; t_i \in Cc(T_i))$

3.1.3 Implementation

In the domain for study of types, we need to store:

- the type of each term and each subterm of the abstract substitution, (*Ty* component), note that types are, here, elements of the generalized mode domain as presented in the chapter aboute types and in [20],
- the form of each term and the form of each subterm if defined, i.e. their functor, their arity and the list of their subterms, (*frm* component),
- which terms have the same value (same type and same form), if there are some (*sv* component),
- and for each pair of subterms, whether they may share variables together or not, (*ps* component);

- we can also store $ps*$, the transitive closure of ps; it is discussed later in this chapter.

In Pascal, the abstract substitutions were implemented as records. But, first of all, as Pascal needs static declarations we have to define the maximum size of each structure. So we define a few constants[3] and then the data types, as shown on figure 3.2

The field *bottom* is added for ease of use. It tells whether the abstract substitution is $\perp$ or not. If an abstract substitution is $\perp$ then the boolean *bottom* is set to *true* and all other fields are undetermined. Else, *bottom* is set to $false$ and all fields must be correctly set up, that is $sizesv$ and $sizefrm$ are well defined, as well as $sv[1] \ldots sv[sizesv]$, $frm[1] \ldots frm[sizefrm]$ and $ps[1,1] \ldots ps[sizefrm, sizefrm]$, other elements of these arrays being undetermined. For an element $frm[i]$ to be well defined, both fields *mode* and $form$ must be well defined as well as, if $form$ is set to *true* (that means the form of the subterm is defined), $functor$, $arity$ and $arg[1] \ldots arg[arity]$.

For example of use of that data type, let us code the following abstract substitution:

$$\{NGV(1) : f(Var(2), Var(2), Any(3)), Ground(4) : a,$$
$$NGV(1)f : (Var(2), Var(2), Any(3))\}, \quad ps = \{(2,2),(3,3)\}$$

This substitution contains three terms, $sizesv = 3$, and four subterms, $sizefrm = 4$. The subterms are coded the following way:

1. $mode = NGV$, $form = true$, $functor = f$, $arity = 2$, $args[1] = 2$, $args[2] = 2$
2. $mode = Var$, $form = false$
3. $mode = Any$, $form = false$
4. $mode = Ground$, $form = true$, $functor = a$, $arity = 0$

Since the first and the last terms are the first subterm and since the second term is the fourth subterm, $sv = (1,4,1)$. And, finally,

$$ps[i,j] = false, 1 \leq i,j \leq 4, \text{ excepted } ps[2,2] = ps[3,3] = true$$

When the implmentation is done in C language, we can take advantage of the dynamic allocation of variables, at the cost of ease of use due to manipulation of a lot of pointers. Note too that *psclose* ($ps*$) disappears in the C version as explained in the next chapter.

In C language, there is no boolean type, but it can be emulated with others. The choice of the type to use is very important. Indeed, C language can handle

[3]note that the values given here are the best suited for all the tests, though it might be necessary to change them. For instance, to run the program on `CS` (see chapter about results), constant 'maxfrm' must be set to 100

bytes and bits of the bytes too. So it could be a good idea to use each bit of a byte as a different boolean value. But, to read or to write a particular bit in a byte involves some computations that do not exist when a byte is used in its whole for a single boolean value. Thus, using bits leads to gain some memory, but to lose time in comparison with bytes. Both methods have been implemented. With bytes for better results in time for computers which have a lot of memory. With bits for smaller computers, when computation time is not the main matter of the user. All the results of the optimizations are given with respect to both methods.

The declarations in C language are shown in figure 3.3 and its graphical representation is shown on figure 3.4. Note that *teltps* must be define as **int** to implement booleans with bits and as **chars** to implement booleans with bytes. *psblocksize* is the number of boolean values stored in one element of *ps*. A **char** is for one boolean value (*psblocksize* = 1) and an **int** is for a group of 16 or 32 boolean values (*psblocksize* = 16 or 32) depending of the machine on which the program is run. Note too that a field *psclose* is present in the Pascal version and not in the C version. This is explained in the next chapter.

To create a new abstract substitution is very simple in Pascal, just declare a variable as pointer $\hat{}\,tas$, perform a **new** on that variable and use it. In C, it is much more complicated. Indeed, an abstract substitution is made of several parts: the basic part with *bottom*, *sizesv*, *sizefrm*, pointers to *sv* and *frm* and *ps*, the *sv* part and the *frm* part that is a set of pointers to each form. Since *ps* vary in length from one substitution to the other, its size must be calculated for each new substitution as $sizefrm^2$ elements. Then, the size of the basic part is computed as

$$\textbf{sizeof}(tas) + (sizefrm^2 - 1)/(psblocksize/\textbf{sizeof}(teltps))$$

where '-1' is due to the fact that the first element of *ps* is already taken into account with '**sizeof**(*tas*)' (*teltps ps*[1]) and *psblocksize*/**sizeof**(*teltps*) is the number of boolean values stored in one byte (1/1 = 1 with **char**, 32/4 = 8 with **int**). A first memory allocation of that size is done. Then, a second allocation of size

$$sizesv * \textbf{sizeof(int)}$$

is done for the *sv* part. A third allocation of size

$$sizety * \textbf{sizeof}(*tfrm)$$

must be done for the list of pointers to *frm* components. And finally, one more allocation is necessary for each *frm* component. These last ones are of size

$$\textbf{sizeof}(tfrm) + (arity - 1) * \textbf{sizeof(int)}$$

3.2 The *mode* domain

The *mode* domain focuses interest on the mode of variables. An abstract substitution is implemented with three components:

- the mode
- the same value component to express that two variables have the same value.
- the possible sharing.

Although next sections present a more complete description of these components, the reader can find an exhaustive specification in [20, II/13].

3.2.1 Formal Presentation

Modes

An *abstract term* is characterized by its mode namely: $ground, var$ and any. Let us define the concretization function which establishes the link between abstract and concrete universe. Let $E = \{ground, var, any\}$ be the set of modes. Cc is defined as:

$$\begin{aligned} Cc :\ & E \longrightarrow P(T) \\ & ground\{\ t : t \text{ is a basic term}\} \\ & var \quad \{\ v : v \text{ is a variable}\} \\ & any \quad T \end{aligned}$$

where T is the set of all the Prolog terms and $P(T)$ is the set of all the possible partitions of T. E is endowed with a relation $\leq$ defined as: $ground \leq var \leq any$.

A *mode substitution* defined on the set of variables $D = \{X_1, \ldots, X_n\}$ is of the following form:

$$\{X_1 \leftarrow m_i, \ldots, X_n \leftarrow m_n\}$$

We define the set $S_m(D)$ of all the mode substitutions as

$$S_m(D) = \{\{X_1 \leftarrow m_1, \ldots, X_n \leftarrow m_n\} \mid m_i \in E\} \cup \{\bot\}$$

The concretization function is defined as

$$Cc(\alpha) = \begin{cases} \emptyset & \text{if } \alpha = \bot \\ \{\{X_1 \leftarrow t_i\}_i \mid t_i \in Cc(m_i)\} & \text{otherwise} \end{cases}$$

$S_m(D)$ is endowed with a partial order relation $\leq$ defined as:

$$\begin{aligned} \forall \alpha, \beta \in S_m(D) : \alpha \leq \beta \Leftrightarrow & \\ & \alpha = \bot \text{ or} \\ & \alpha = \{X_i \leftarrow m_i\}_i \text{ and } \beta = \{X_i \leftarrow p_i\}_i \text{ and } m_i \leq p_i \ \forall i \end{aligned}$$

Same Value

The *same value* component may be defined the same way as for the *type* domain.

Possible Sharing

The *possible sharing* (*ps*) expresses that two abstract terms may eventually share a variable. In other words, when there is no "possible sharing" between two asbtract terms, these ones can not share any variable. *ps* has a meaning only for abstract terms which are not *ground*. *ps* is an element of PS defined as:

$$PS(D) = \{X : X = P(Y) \wedge Y \in D\}$$

where D is the domain of abstact terms.
The concretization function is defined as:

$$\begin{array}{ll} Cc: PS(D) \longrightarrow & CS_D \\ \quad ps & \{\{X_i \leftarrow t_i\}_i : var(t_i) \cap var(t_j) \neq \emptyset \Rightarrow ps(X_i, X_j)\} \end{array}$$

There is also a partial order relation defined as:

$$\forall ps_1, ps_2 \in PS(D): \; ps_1 \leq ps_2 \Leftrightarrow \forall i, j : ps_2(X_i, X_j) \Rightarrow ps_1(X_i, X_j)$$

Asbtract Substitutions

We have now all the elements to define completely the abstract domain:

$$AS_D = S_m(D) \times P(D) \times PS(D)$$

The concretization is defined as

$$\begin{array}{lll} Cc: AS_D & \longrightarrow & CS_D \\ \quad \beta = (\delta, sv, ps) & & Cc(\delta) \cap Cc(sv) \cap Cc(ps) \end{array}$$

AS_D is endowed with a partial order relation defined as:

$$(\delta, sv, ps) \leq (\delta', sv', ps') \Longleftrightarrow \delta \leq \delta' \wedge sv \leq sv' \wedge ps < leqps'$$

Properties
All the Cc functions are monotonic and $(AS_D, \leq)$ is a cpo.

3.2.2 Implementation

In the domain for the study of modes, we need to store:

- the mode of each variable in the domain of the abstract substitution,
- for each pair of terms, whether they have the same value or not,
- for each pair of terms, whether they may share variables together or not.

Figures 3.5 and 3.6 show the implementations of the data type for abstract substitutions respectively in Pascal and in C. Figure 3.7 shows a graphical representation of the C version. The presence of *psclose* in the Pascal version and its absence in the C version is explained in the next section.

To create a new abstract substitution, as in the *type* domain, its easy in Pascal and more complicated in C. But now, substitutions only have three parts: the basic part, the *mode* part and the *sv* part. The second one is simple to obtain, it suffices to allocate a memory area of size

$$size * \textbf{sizeof}(int)$$

Then we could do the same for the *ps* part and we could do the same as with the *type* domain for the basic part. That is, two memory allocations. But their is another solution that makes possible to only do one memory allocation. The solution is to append the *sv* part at the end of the basic part. So, to allocate memory, we must do a request of size

$$\textbf{sizeof}(tas) + (size^2 * 2 - 1)/(psblocksize/\textbf{sizeof}(teltps))$$

where $size^2$ is the number of element in one array, '$*2$' means that both arrays are the same size, '-1' is for $ps[1]$ that already appears in **sizeof**(*teltps*) and (*psblocksize* / **sizeof**(*teltps*)) is the number of boolean values stored in one byte. Then, in order to be able to use the *sv* part, we have to calculate its address:

$$*sv = \textbf{sizeof}(tas) + (size^2 - 1)/(psblocksize/\textbf{sizeof}(teltps))$$

Each time we need to store something for each pair of terms or subterms, we must store, assuming there are n terms or subterms, n^2 boolean values. It leads to very big arrays, memory consuming.

In Pascal, as the size of the arrays must be known at compile time, it is mandatory to estimate a maximum size for them. They must be big enough to store the biggest object the program can handle during execution, and they must all have the same size, even if they are empty. Furthermore, Pascal can not use just one bit for a boolean value, it always use at least one byte. Let us assume, for instance, that the biggest abstract substitutions are 50 subterms long and that booleans are coded on one byte. Then, a simple boolean array is $50 * 50 = 2500$ bytes long. There are at least one of them for each substitution. There can be several thousands of substitutions created during one execution of the program. What a waste! In C language, dynamic storage is permitted, i.e. arrays are just the size they really need to be.

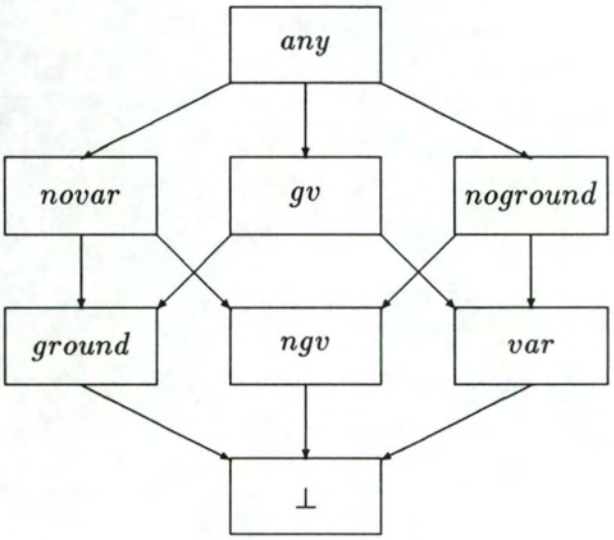

Figure 3.1: Modes for the *type* domain

```
const
   maxsv = 50;
   maxfrm = 50;
   maxarity = 30;

type
   mdg = (Bottom, Ground, Var, NGV, NoGround, NoVar, GV, Any);
   pfunctor = pointer; {a pointer to a structure representing a functor}
   tfrm = record
      mode : mdg;
      form : boolean;
      functor : pfunctor;
      arity : integer;
      args : array[1..maxarity]of integer
   end;
   tas = record
      bottom : boolean;
      sizesv : integer;
      sv : array[1..maxsv]of integer;
      sizefrm : integer;
      frm : array[1..maxfrm]of tfrm;
      ps : array[1..maxfrm,1..maxfrm]of boolean
      psclose : array[1..maxfrm,1..maxfrm]of boolean
   end;
```

Figure 3.2: Data types of abstract substitutions for *type* domain in Pascal

```
#define psblocksize .../* 1 or 32 depending on teltps */

typedef short bool /* short or anything else */
typedef enum {Bottom, Ground, Var, NGV, NoGround, NoVar, GV, Any} mdg
typedef ...teltps /* '...' is either int or char */
typedef struct {
    int arity;
    mdg mode;
    bool form;
    pfunctor functor;
    int args[1];
} tfrm;
typedef struct {
    bool bottom;
    int sizesv;
    int *sv;
    int sizefrm;
    tfrm *frm;
    teltps ps[1];
} tas;
```

Figure 3.3: Data types of abstract substitutions for *type* domain in C

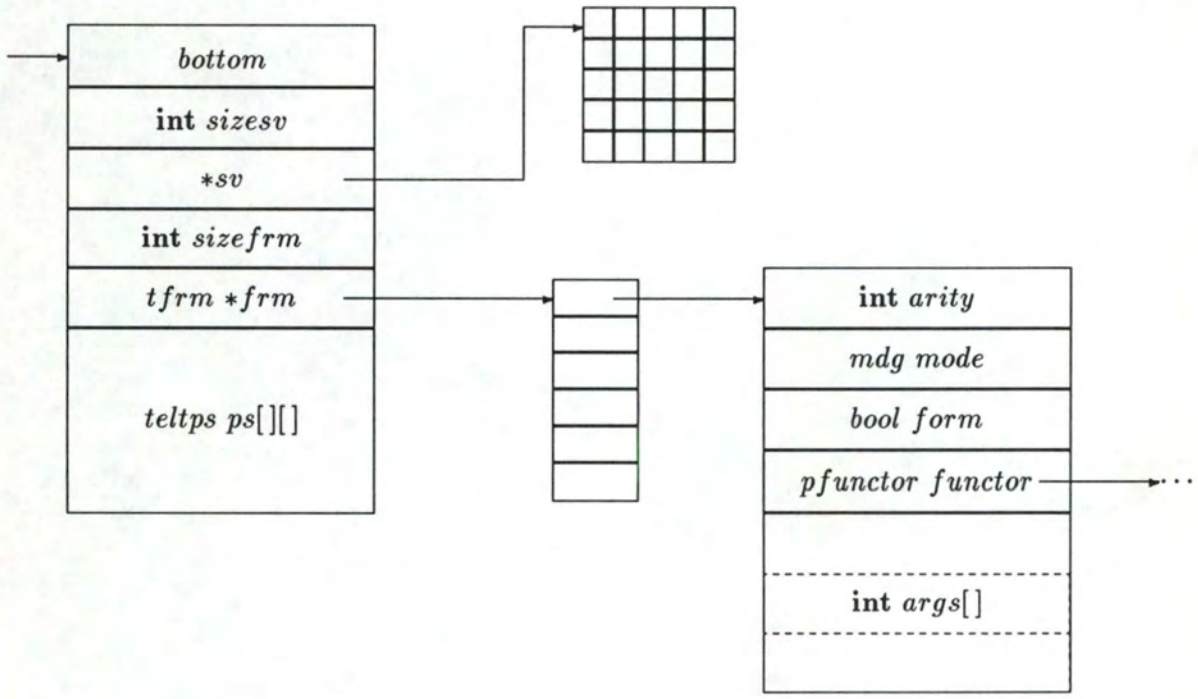

Figure 3.4: Abstract substitutions for the *type* domain

```
const
   maxsv = 50;

type
   md = (Bottom, Ground, Var, Any, Top);
   tas = record
      size : integer;
      modes : array[1..maxsv]of md;
      sv : array[1..maxsv,1..maxsv]of boolean
      ps : array[1..maxsv,1..maxsv]of boolean
      psclose : array[1..maxsv,1..maxsv]of boolean
   end;
```

Figure 3.5: Data types of abstract substitutions for *mode* domain in Pascal

```
#define psblocksize .../* 1 or 32 depending on teltps */

typedef enum {Bottom, Ground, Var, Any, Top} md
typedef ...teltps /* '...' is either int or char */ typedef struct {
   int size;
   md *modes;
   teltps *sv;
   teltps ps[1];
} tas;
```

Figure 3.6: Data types of abstract substitutions for *mode* domain in C

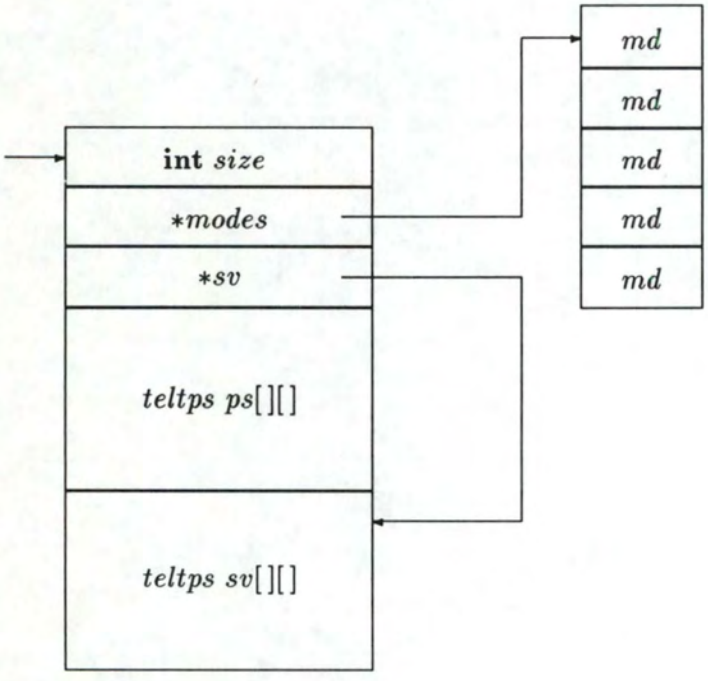

Figure 3.7: Abstract substitutions for the *mode* domain

Chapter 4

The original program

We have presented so far what abstract interpretation is and the domains on which we will use it. In this chapter, we present the basic abstract interpretation algorithm and its implementation. The algorithm was first described in [13]. A first implementation is presented in [15]. It is written in Pascal. It has been translated in C and a few optimizations have been done. We shall present both versions and all the optimizations.

4.1 Goal Dependencies

In the first chapter, we defined all the operations necessary to design the algorithm. However, one of the main concerns in its design has been the detection of redundant computations. They may occur in a variety of situations. For instance, the value of a pair (α, q) may have reached its definitive value (the value of (α, q) is in the least fixpoint) and hence subsequent considerations of (α, q) should only look up its value instead of starting a subcomputation. Another important case (especially in logic programming) are mutually recursive programs. For those programs, we would like the algorithm to reconsider a pair (α, q) only when some elements which it is depending upon have been updated. In other words, keeping track of the goal dependencies may substantially improve the efficiency on some classes of programs.

The algorithm includes specific data structures to maintain goal dependencies. We only introduce the basic notions here. They have been borrowed to [13].

Definition 7 A dependency graph is a set of tuples of the form $\langle(\beta, p), lt\rangle$ where lt is a set $\{(\alpha_1, q_1), \ldots, (\alpha_n, q_n)\}$ $(n \geq 0)$ such that, for each (β, p), there exists at most one lt such that $\langle(\beta, p), lt\rangle \in dp$.

We denote by $dp(\beta, p)$ the set lt such that $\langle(\beta, p), lt\rangle \in dp$ if it exists. We also denote by $dom(dp)$ the set of all (β, p) such that $\langle(\beta, p), lt\rangle \in dp$ and by $codom(dp)$ the set of all (α, q) such that there exists a tuple $\langle(\beta, p), lt\rangle \in dp$ satisfying $(\alpha, q) \in lt$.

The basic intuition here is that $dp(\beta, p)$ represents at some point the set of pairs which (β, p) depends directly upon. To be complete, we need to define the transitive closure of the dependencies.

Definition 8 Let dp be a dependency graph and assume that $(\beta, p) \in dom(dp)$. The set $trans\_dp(\beta, p, dp)$ is the smallest subset of $codom(dp)$ closed by the two following rules:

1. if $(\alpha, q) \in dp(\beta, p)$ then $(\alpha, q) \in trans\_dp(\beta, p, dp)$;
2. if $(\alpha, q) \in dp(\beta, p)$, $(\alpha, q) \in dom(dp)$, and $(\alpha', q') \in trans\_dp(\alpha, q, dp)$ then $(\alpha', q') \in trans\_dp(\beta, p, dp)$.

Now $trans\_dp(\beta, p, dp)$ represents all the pairs which, if updated, would require reconsidering (β, p). (β, p) will not be reconsidered unless one of these pairs is updated.

We are now in position to specify the last three operations needed to present the algorithm:

- `REMOVE_DP`$(modified, dp)$, where $modified$ is a list of pairs $(\alpha_1, p_1), \ldots, (\alpha_n, p_n)$ and dp is a dependency graph, removes from the dependency graph all elements $\langle(\alpha, q), lt\rangle$ for which there is a $(\alpha_i, q_i)) \in trans\_dp(\alpha, q, dp)$.
- `EXT_DP`(β, p, dp) inserts an element $\langle(\beta, p), \emptyset\rangle$ in dp;
- `ADD_DP`$(\beta, p, \alpha, q, dp)$ simply updates dp to include the dependency of (β, p) wrt (α, q) (after its execution $(\alpha, q) \in dp(\beta, p)$.

The algorithm makes sure that the elements (β, p) that need to be reconsidered are such that $(\beta, p) \notin dom(dp)$.

4.2 The generic abstract algorithm

We are now in position to present the generic abstract interpretation algorithm ([13]). The algorithm is composed of three procedures and is shown in Figure 4.1.

The top-level procedure is Procedure `solve` which, given an input substitution β_{in} and a predicate symbol p, returns the set of abstract tuples sat containing $(\beta_{in}, p, \beta_{out})$ belonging to the least fixpoint[1] and the final dependency graph. Given the results, it is straightforward to compute the set of pairs (α, q) used by (β, p), their values in the fixpoint, as well as the abstract substitutions in any program point.

Procedure `solve_call` receives as inputs an abstract substitution β_{in}, its associated predicate symbol, a set $suspended$ of pairs (α, q), sat, and a dependency graph

[1] In the case of infinite domains, it is possible that the algorithm returns an upper approximation to the least fixpoint due to our use of widening operations to guarantee termination.

```
procedure solve(in β_in, p; out sat, dp)
begin
   sat := ∅;
   dp := ∅;
   solve_call(β_in, p, ∅, sat, dp)
end

procedure solve_call(in β_in, p, suspended; inout sat, dp)
begin
   if (β_in, p) ∉ (dom(dp) ∪ suspended) then
   begin
      if (β_in, p) ∉ dom(sat) then
         sat := EXTEND(β_in, p, sat);
      repeat
         β_out := ⊥;
         EXT_DP(β_in, p, dp);
         for i := 1 to m with c_1, ..., c_m clauses-of p do
         begin
            solve_clause(β_in, p, c_i, suspended ∪ {(β_in, p)}, β_aux, sat, dp);
            β_out := UNION(β_out, β_aux)
         end;
         (sat, modified) := ADJUST(β_in, p, β_out, sat);
         REMOVE_DP(modified, dp)
      until (β_in, p) ∈ dom(dp)
   end
end

procedure solve_clause(in β_in, p, c, suspended; out β_out; inout sat, dp)
begin
   β_ext := EXTC(c, β_in);
   for i := 1 to m with b_1, ..., b_m body-of c do
   begin
      β_aux := RESTRG(b_i, β_ext);
      switch (b_i) of
      case X_j = X_k:
         β_int := AI_VAR(β_aux)
      case X_j = f(...):
         β_int := AI_FUNC(β_aux, f)
      case q(...):
         solve_call(β_aux, q, suspended, sat, dp);
         β_int := sat(β_aux, q);
         if (β_in, p) ∈ dom(dp) then
            ADD_DP(β_in, p, β_aux, q, dp)
      end;
      β_ext := EXTG(b_i, β_ext, β_int)
   end;
   β_out := RESTRC(c, β_ext)
end
```

Figure 4.1: The Generic Abstract Interpretation Algorithm

dp. The set *suspended* contains all pairs (α, q) for which a subcomputation has been initiated and not been completed yet. The procedure is responsible to consider (or reconsider) the pair (β_{in}, p) and to update *sat* and *dp* accordingly. The core of the procedure is only executed when (β_{in}, p) is not suspended and not in the domain of the dependency graph. If (β_{in}, p) is suspended, no subcomputation should be initiated. If (β_{in}, p) is in the domain of the dependency graph, it means that none of the elements which it is depending upon have been updated. Otherwise, a new computation with (β_{in}, p) is initiated. This subcomputation may extend *sat* if it is the first time β_{in} is considered. The core of the procedure is a `repeat` loop which computes the best approximation of (β_{in}, p) given the elements of the suspended set. Local convergence is attained when (β_{in}, p) is in the domain of the dependency graph. One iteration of the loop amounts to executing each of the clauses defining p and computing the union of the results. If the result produced is greater or not comparable to the current value of (β_{in}, p), then the set of abstract tuples is updated and the dependency graph is also adjusted accordingly. Note that the call to the clauses is done with an extended suspended set since a subcomputation has been started with (β_{in}, p). Note also that, before executing the clauses, the dependency graph has been updated to include (β_{in}, p) (which is guaranteed not to be in the domain of the dependency graph at that point). (β_{in}, p) can be removed from the domain of the dependency graph during the execution of the loop if a pair which it is depending upon is updated.

Procedure `solve_clause` executes a single clause for an input pair and returns an abstract substitution representing the execution of the clause on that pair. It begins by extending the substitution with the variables of the clause, then executes the body of the clause, and terminates by restricting the substitution to the variables of the head. The execution of a goal requires three steps:

- restriction of the current substitution β_{ext} to its variables giving β_{aux};
- execution of the goal itself on β_{aux} producing β_{int};
- propagation of its result β_{int} on β_{ext}.

If the goal is concerned with unification, the operations `AI_VAR` and `AI_FUNC` are used. Otherwise, procedure `solve_call` is called and the result is looked up in *sat*. Moreover, if (β_{in}, p) is in the domain of the dependency graph, it is necessary to add a new dependency. Otherwise, it means that (β_{in}, p) needs to be reconsidered anyway and no dependency must be recorded.

4.3 Foundations

The foundations are the true results of the abstract interpretation of a Prolog program P. They are sets of tuples (β, p, β') that extracted from the *sats*. These elements are those for which the input abstract substitution is the abstraction of a

substitution occuring during the execution of the analysed program. Indeed, when analysing a program, approximated results due to suspended recursive calls can lead to a number of tuples that would not appear if it was possible to obtain directly the right result. Those tuples are "noise" in the *sat*s and must be ignored in the calculation of foundation. In this section, we define the foundations more precisely and we explain the algorithm used to calculate them.

4.3.1 Definition

Let us now define the foundations, as presented in [15].

Definition 9 The pair (β, p) is based in *sat* iff $\forall sat'$ such that sat' is total and $sat \subseteq sat'$, the computation of $T_p(\beta, p, sat')$ does not require values of sat' not belonging to *sat*. In that case, and in that case only, $base(\beta, p, sat)$ is the smallest set D such that (β, p) is based in $sat\ |_D$.

Definition 10 The pair (β, p) is founded in *sat* iff $\exists D : (\beta, p) \in D$ and $\forall (\alpha, q) \in D$: (α, q) is based in $sat\ |_D$. In that case, and in that case only, $foundation(\beta, p, sat)$ is the smallest set D such that (β, p) is founded in $sat\ |_D$.

For instance, let us assume $p =$ `APPEND`$/3$ is studied with $\beta_{in} = \{X_1 \mid Var, X_2 \mid Var, X_3 \mid Gro\}$. The only resulting *sat* will only contain one tuple:

$$(\{X_1 \mid Var, X_2 \mid Var, X_3 \mid Gro\}, \texttt{APPEND}/3, \{X_1 \mid Gro, X_2 \mid Gro, X_3 \mid Gro\})$$

Let D be a set containing the only pair (β_{in}, p). Thus, $sat\ |_D = sat$. The pair (β_{in}, p) is based in *sat* because its evaluation calls only itself. It is founded in *sat* because the only element of D ((β_{in}, p) itself) is based in $sat\ |_D$. And D is the smallest set such that (β_{in}, p) is founded in $sat\ |_D$ because, otherwise, D must be empty, so is $sat\ |_D$ and nothing can be founded in the empty set. Hence, the foundation of $(\beta_{in}, \texttt{APPEND}/3)$ is that set D. That is, during execution of `APPEND`$/3$ with a substitution for which β_{in} is an abstraction, all calls of `APPEND`$/3$ are done with substitutions for which β_{in} is an abstraction too.

4.3.2 The algorithm for the calculation of the foundations

Let P, a program, be a set of predicates $\{p_i/m_i\}$, each of them being a set of clauses of the form:

$$C \equiv H(X_1, X_2, \ldots, X_m) : -B_1, B_2, \ldots, B_p$$

where H is the head and B_k are subgoals (built-ins or procedure calls). Let us assume that the predicate p/m of P is called with β and that subgoal B_k leads to a recursive (locally or mutually) call of p/m. When this recursive call occurs, its execution is avoided and its best result so far, which is stored in the corresponding *sat*, is returned. All $B'_k, k' > k$, are then examined with an input substitution depending on

that result. Let $\{\beta^1_{k\ out}, \ldots, \beta^l_{k\ out}\}$ be a set of different outputs of B_k at subsequent calls. Let us assume that $B_{k'}$, $k' > k$, is a procedure call. It is examined, at the i^{th} call, with an input substitution depending on $\beta^i_{k\ out}$, say $\beta^i_{k'in}$. Thus, the series of calls to p/m will produce a list of tuples $(\beta^1_{k'in}, B_{k'}, \beta^1_{k'out}), \ldots, (\beta^l_{k'in}, B_k, \beta^l_{k'out})$ which all appear in the Hasse diagram of p/m.

But, in fact, only the last one, $(\beta^l_{k'in}, B_k, \beta^l_{k'out})$, is correct. All the previous ones are present in the *sat*s because of the suspension of the execution of the recursive calls, but they do not have been improved and their result is false.

So, the algorithm for the calculation of the foundations is very similar to the abstract interpretation algorithm. It suffices to run it a second time with the *sat*s calculated the first time because they contain the best approximation possible for each pair (β_i, p_i). Hence, the good result is directly known and there is no more reexamination in order to imrpove it and no more calculation of the `UNION`. Each visited pair is part of the foundation. It is shown on figure 4.2.

4.4 The original program

A first implementation of the program has been written by Pascal Van Hentenryck. It is described in [15]. The purpose of our work is to improve that program. Thus, as it is a basis for the following, we need to understand it before working on it. This section will describe some interesting aspects of the implementation of the program to know for a better understanding of the optimizations.

This first implementation was written in PASCAL. But, it allows only static allocation of memory. So it has been translated in C language, which allows dynamic allocation of memory. In a first time, we will describe the most important data structures used. From these, we can make some choices in order to minimize memory size used by the program or to minimize the execution time. Then, we will explicit the design of the program.

4.4.1 Abstract substitutions

Abstract substitutions are the most important structures manipulated by the program. They contain a lot of information and they are the most used data structures. Thus, it is important to write them carefully. Two different abstract domains are implemented. Each of them has its own abstract substitutions, different one from the other. We already described their implementation in the previous chapter.

4.4.2 Transitive closure of Possible Sharing

In the previous chapter, when we described the implementation of the abstract substitutions, we underlined that the *ps** component was included in the Pascal version (*psclose* field), but not in the C version. That is due to an interesting optimization optimization. We now desbribe that improvement.

```
procedure foundation(in β_in, p, sat; out foundation)
begin
   foundation := ∅;
   foundation_call(β_in, p, ∅, sat, dp, foundation)
end

procedure foundation_call(in β_in, p, suspended, sat; inout foundation)
begin
   if (β_in, p) ∉ (suspended ∪ foundation) then
   begin
      foundation := foundation ∪ {(β_in, p, sat(β_in, p))};
      for i := 1 to m with c_1, ..., c_m clauses-of p do
         foundation_clause(β_in, p, c_i, suspended ∪ {(β_in, p)}, sat);
   end
end

procedure foundation_clause(in β_in, p, c, suspended, sat)
begin
   β_ext := EXTC(c, β_in);
   for i := 1 to m with b_1, ..., b_m body-of c do
   begin
      β_aux := RESTRG(b_i, β_ext);
      switch (b_i) of
      case X_j = X_k:
         β_int := AI_VAR(β_aux)
      case X_j = f(...):
         β_int := AI_FUNC(β_aux, f)
      case q(...):
         foundation_call(β_aux, q, suspended, sat, dp);
         β_int := sat(β_aux, q);
      end;
      β_ext := EXTG(b_i, β_ext, β_int)
   end
end
```

Figure 4.2: The calculation of the foundation

$$ps = \{(t_i, t_j) \mid t_i \text{ may share one or more variable with } t_j\}$$

A more complete definition can be found in [20].

The transitive closure $ps*$ of ps is calculated with the following rules:

- $\forall (t_i, t_j) \in ps, (t_i, t_j) \in ps*$
- $\forall i, j, k, (t_i, t_j) \in ps*$ and $(t_j, t_k) \in ps* \Longrightarrow (t_i, t_k) \in ps*$
- $t_i = f(\ldots, t_j, \ldots) and (t_k, t_j) \in ps \Longrightarrow (t_i, t_k) \in ps$

In the original program, both ps and $ps*$ are stored in memory. When a procedure that modifies ps is over, $ps*$ is recalculated. By that way, it is at disposal of all procedures that need it.

The main goal we have in sight when translating the program from Pascal to C is to run it with less memory, in order to be able to use it with bigger Prolog programs. So, a way of doing this is not to store $ps*$ any more, but to calculate it only when necessary.

The way ps and $ps*$ are implemented is the following: for a substitution made of n terms and subterms, there are n^2 booleans, one for each pair $(t_i, t_j), 1 \leq i, j \leq n$ telling whether $(t_i, t_j) \in ps$ or not.

For example, let us suppose we have a substitution made of 20 terms (it is common in all previous tests). And let us suppose, with search for speed in sight, ps is implemented with *chars* to store the boolean values. Then it takes $20^2 = 400$ bytes of memory per substitution for ps as well as for $ps*$. If more than 100 substitutions are created, we can really gain a lot of place by avoiding to store $ps*$.

In fact, this little trick is much more interesting. Indeed, the program now computes $ps*$ much less times. With the first method, a $ps*$ that is used several times is computed only once, but there are also a lot of computation that are never used. If we measure the time necessary for one test with both methods, we can note a very big difference. In fact, this improvement is responsible of a large part of the percentage of the time won with *prefix* as well as with *caching*.

4.4.3 *Sats*

All the abstract substitutions are elements of *sats*. These *sats* take the form of Hasse diagrams. But it is worth having a complete section to describe these diagrams in detail, so it will be discussed later on.

4.4.4 The dependency graph

Another important data structure is the dependency graph described in the begining of this chapter. It is not important in the sense of its size because it just contains a

few references, just a few bytes, but in the sense of its utility. A bad implementation of this graph could lead to very big amount of unnecessary computations as we shall explain.

According to [15], the dependency graph is a set of pairs composed of a pair (β, p) and a set lt of pairs (α_i, q_i) indicating that (β, p) must be reconsidered if and only if at least one (α_i, q_i) has been reconsidered. This was presented that way for facility and beauty. But, it is more efficient to implement it in the opposite way, namely, elements are a set lt' of pairs (α_i, q_i) associated to a pair (β, p) where each (α_i, q_i) is to be reconsidered if and only if (β, p) is. Let us note $dp'(\beta, p)$ that set lt'.

Let us explain why the reverse method is the easiest.

Firstly, the creation of the graph. Procedure ADD_DP$(\beta, p, \alpha, q, dp)$, as described previously, adds a dependency (α, q) to $dp(\beta, p)$. But, to do that or to add (β, p) to $dp(\alpha, q)$ is exactly the same. So, no gain and no lost in creating the graph in reverse sense.

Then, the use and the destruction of the graph. When the consideration of the pair (β, p) is over, all the pairs depending on it must be reconsidered. So, we need to find all these pairs and mark them to 'to be reconsidered'. With the first method, it is necessary to look at every pair $(\alpha, q) \in dom(dp)$ and to check if $(\beta, p) \in dp(\alpha, q)$. All pairs (α, q) verifying that condition must be marked. But we don't only need to search all (α, q) such that $(\beta, p) \in dp(\alpha, q)$, we must mark all (α, q) such that $(\beta, p) \in trans\_dp(\alpha, q, dp)$, so we must redo the same recursively for each (α, q) found. Note that there would be an infinite loop if $\exists(\alpha, q)$ such that $(\alpha, q) \in trans\_dp(\alpha, q, dp)$. To avoid this problem, the searched elements are removed from the graph when found, that is, if $(\beta, p) \in dp(\alpha, q)$ then $dp(\alpha, q) := dp(\alpha, q) \backslash \{(\beta, p)\}$. On the other hand, with the second method, it suffices to mark all elements of $dp'(\beta, p)$ and redo the same recursively for all these elements. Note that we still need to remove the marked elements from the graph, that is $dp'(\beta, p) = \emptyset$ when all its elements are marked and recursively examined. Hence, with the first method, dp in its whole must be examined, possibly several times, but with the second method, only $trans\_dp(\beta, p, dp')$ must be examined. The theorical complexity is $O(n^2)$ in the first case and $O(n)$ in the second case. In case of a big program with few dependencies, the ratio of number of examined elements may be of several thousands to one.

4.4.5 The implementation

The program of abstract interpretation runs in three phases. Firstly, a phase of normalization of a Prolog program. Secondly, the abstract interpretation algorithm. And lastly, the calculation of the foundation.

As described in a previous section, the program works with normalized Prolog programs. But it is not very convenient for the user to write a program in a normalized way. So, the first phase during the execution of the program is a translation of a standard Prolog program into its normalized form.

The second part is the most important part of the execution. It is an implementation of all the algorithms and functions given so far, as well as the Hasse diagram explain in the next section.

The implementation is done in a top-down fashion with different levels of procedures:

- First level: the generic algorithm: procedures `solve`, `solve_call` and `solve_clause`
- Second level: all abstract operations used by the first level: `UNION`, `AI_VAR`, `AI_FUNC`,...
- Third level: a series of sophisticated functions for manipulation of data structures, that is functions for manipulation of data structures like abstract substitutions, Hasse diagrams,... regarding their semantics; for instance, comparison of two substitutions (are they $=$, $<$, $>$ or not comparable?), unification of substitutions, search of an element in Hasse diagram,...
- Fourth level: a series of functions for manipulation of data structures without regard to their semantics, such as syntaxical comparison of abstract substitutions (are they equal byte per byte?), copy of an abstract substitution into another,...

The implementation is modular, that is all procedures are gathered in different files in function of their purpose. For instance, the three procedure of the abstract interpretation algorithm are together in one file. All procedures concerning manipulation of the Hasse diagrams (`EXTEND`, `SEARCH`,...) are in another file. A third file is a set of procedures of the third level for manipulation of abstract substitutions (unification, comparison,...). A fourth one contains all the little tools for manipulation of abstract substitutions (assigning, copying, printing, syntaxical comparison,...).

And finally comes the calculation of the foundation which is a simple implementation of the algorithm shown in a previous section.

4.4.6 Abstract Unification

The function `uact1`(β, i, j) unifies two undefined patterns ($frm(i)$ and $frm(j)$) inside an abstract substitution (β). Taking advantage of the fact that `uact1` can modify directly its argument without creating a new substitution, the computation of the new ps component was improved. The old algorithm was composed of three subprocedures:

1. initialization of the result with values by default
2. computation of the new modes (Ty component)
3. computation of the new ps component

This last point was itself composed of three parts:

- initializing the new ps to $false$ ($ps_{new}(i,j) = false\ \forall i,j$)
- transferring the old ps to the new one as:

$$\forall k,l \quad \left.\begin{array}{l} ps_{old}(k,l) \text{ and} \\ Ty_{new}(k) \neq ground \text{ and} \\ Ty_{new}(l) \neq ground \end{array}\right\} \Longrightarrow ps_{new}(k,l)$$

- updating the new ps to take into account the following property:

$$\forall k,l \quad \left.\begin{array}{l} ps_{old}(k,i) \vee ps_{old}(k,j) \text{ and} \\ ps_{old}(l,i) \vee ps_{old}(l,j) \text{ and} \\ Ty_{new}(i) \neq ground \end{array}\right\} \Longrightarrow ps_{new}(l,k)$$

In the new implementation, we work directly on β_{old} ($\beta_{old} \equiv \beta_{new}$). Consequences are that:

- subprocedure 1 can disapear
- subprocedure 2 updates also directly the ps component such that

$$ps(l,k) = false \text{ if } (Ty_{new}(l) = ground \ \vee \ Ty_{new}(k) = ground)$$

- subprocedure 3 computes only two things:

$$\forall k,l : k,l \notin \{i,j\} \quad \left.\begin{array}{l} ps_{old}(k,i) \vee ps_{old}(k,j) \text{ and} \\ ps_{old}(l,i) \vee ps_{old}(l,j) \text{ and} \\ Ty_{new}(i) \neq ground \end{array}\right\} \Longrightarrow ps_{new}(l,k)$$

$$\forall k \ \ ps_{old}(i,k) \vee ps_{old}(j,k) \Leftrightarrow ps_{new}(i,k), ps_{new}(k,i), ps_{new}(j,k), ps_{new}(k,j)$$

We have eliminated so a double loop and a heavy initialization.

4.5 Hasse diagrams

As described previously, function `EXTEND` extends a set of abstract tuples. There is one set for each $p \in P$, say sat_p. Those sets are very important because they are the ones which contain all the informations obtained by execution of the program.

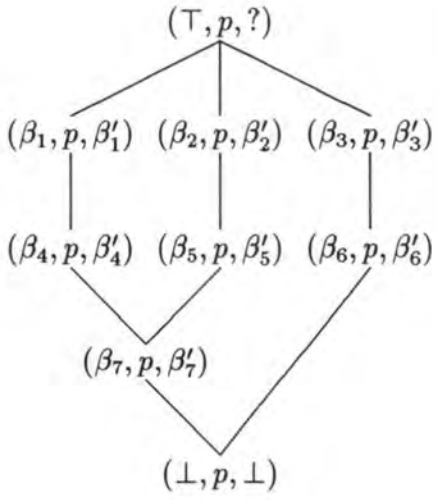

Figure 4.3: An example of Hasse diagram

4.5.1 Definition

These *sats* are organized as *Hasse diagrams*. There is one diagram for each predicate p/n. They are organized in function of the abstract substitution β_{in}. Two elements of a dirgram (β_1, p, β'_1) and (β_2, p, β'_2) are linked together if and only if one is the father of the other.

Definition 11 (β_1, p, β'_1) is a father of (β_2, p, β'_2), if

- $\beta_1 \sqsubseteq \beta_2$
- there is no β_3 such that (β_3, p, β'_3) exists and $\beta_1 \sqsubseteq \beta_3$ and $\beta_3 \sqsubset \beta_2$

Definition 12 Similarly, (β_1, p, β'_1) is a son of (β_2, p, β'_2) iff (β_2, p, β'_2) is a father of (β_1, p, β'_1).

Definition 13 Let $sat_p = \{(beta_i, p, \beta'_i) \mid 1 \leq i \leq n\}$ be a *sat* containing n elements and let (β, p, β') be an element of sat_p. Then,

$$Fathers(\beta, p, \beta') = \{(\beta_i, p, \beta'_i) \mid (\beta_i, p, \beta'_i) \text{ is a father of } (\beta, p, \beta'), 1 \leq i \leq n\}$$

$$Sons(\beta, p, \beta') = \{(\beta_i, p, \beta'_i) \mid (\beta_i, p, \beta'_i) \text{ is a son of } (\beta, p, \beta'), 1 \leq i \leq n\}$$

All elements in such a diagram have a common ancestor ($\top$) and a common descendant ($\bot$). This is added for the ease of implementation.

Let us examine, for instance, the Hasse diagram shown on figure 4.3. It means that:

- $\bot < \beta_7 < \beta_4 < \beta_1 < \top$
- $\bot < \beta_7 < \beta_5 < \beta_2 < \top$
- $\bot < \beta_6 < \beta_3 < \top$
- and the elements that are not in relation here above are not comparable, e.g. β_1 and β_2 ca not be compared.

4.5.2 The original search method

Two functions must be performed on these diagrams: their extension, i.e. adding an element to them, and the search for one element. The first one is the operation `EXTEND` we already spoke about. The second is the one we speak about in this section. Note that this function is used by `EXTEND` in order to put the new entry in its suitable place.

Let us name $\beta_1, \ldots, \beta_n$ the input substitutions of the n tuples of the diagram sat_p associated to the predicate p and $(\beta, p, ?)$ the element to search for. The first searching method is to look at all the elements of the diagram in a depth first search way. The search may be over for two reasons:

- we have found i such that $\beta_i = \beta$,
- we are sure $\forall i$, $\beta_i \neq \beta$.

In the later case, we need also to know where $(\beta, p, ?)$ can be placed, for use by function `EXTEND`. That is, we need the sets

$$F = Fathers(\beta, p, ?)$$

$$S = Sons(\beta, p, ?)$$

Figures 4.4 and 4.5 shows the original algorithm for function `SEARCH`(in (β, p), out $found$, (β_i, p, β_i'), F, S) where $found$ is a boolean value that is $true$ if $(\beta, p, ?)$ has been found in sat_p. In that case, the tuple is set up correctly ($(\beta_i, p, \beta_i') = (\beta, p, sat(\beta, p))$) and the sets F and S are undefined. Else, if $found$ is false, the output tuple is undefined and F and S are the sets of all fathers and all sons of $(\beta, p, ?)$. This procedure has a side effect. It suppresses the links between the elements of F and their sons and between the elements of S and their fathers for the ease of the later insertion of the $(\beta, p, ?)$. Note too that the implementation is slightly more complicated to avoid visiting two times the same element.

4.5.3 The new search method

We can see on figure 4.5 that we need to compare substitutions, β_k and β, in both procedures `SEARCH_TOP_DOWN` and `SEARCH_BOTTOM_UP`. It suffices to test if $\beta_k \leq \beta$

```
procedure SEARCH(in (β,p); out found, (β_i,p,β'_i), F, S);
begin
   found:=false;
   if β = ⊤ then begin
      found:=true;
      (β_i,p,β'_i):=(⊤,p,sat(⊤,p));
   end;
   else begin
      F:=∅; S:=∅;
      SEARCH_TOP_DOWN((⊤,p),(β,p),found,F,(β_i,p,β'_i));
      if not found then
         SEARCH_BOTTOM_UP((⊥,p),(β,p),S)
   end
end;
```

Figure 4.4: SEARCH procedure

and if $\beta \leq \beta_k$. If both tests are true, then they are equal. If just the first one is true, then $\beta_k < \beta$. And, if just the second one is true, $\beta_k > \beta$. Else, they are not comparable. Hence, to compare two substitutions, we need two tests and those are very time consuming because of the complexity of the substitutions. The idea of optimization is to reduce the number of tests by doing each time only one of them instead of two.

Let us examine the procedure SEARCH_TOP_DOWN. With only one test, we can know whether $\beta \leq \beta_k$ or not. If yes, we still do not know whether $\beta = \beta_k$ or $\beta < \beta_k$, so we suppose $\beta < \beta_k$ and we go deeper in the diagram to examine it. Else, we do not do anything. Like in the original algorithm, when all the sons of one $\beta_k > \beta$ are smaller than β or not comparable to β, that (β_k, p, β'_k) is added to F. This is procedure SEARCH_TOP_DOWN_AUX shwon on figure 4.6. When the examination of the diagram is over, we still do not know whether β is in the diagram or not. But we have the set F which contains elements greater than β and eventually β itself.

Proposition 14 The set F obtained by the procedure SEARCH_TOP_DOWN_AUX contains either $(\beta, p, ?)$ as only element or all its fathers.

Proof Let us suppose that F contains n elements and that the input substitutions of those elements are $\beta_i, 1 \leq i \leq n$. All these β_i are, by construction, greater or equal to β.

Let us suppose that one of them is equal to β. By definition of the Hasse diagram, there are no two equal β_i, so all the other ones must be greater than β. Let us suppose there exists such another one, say β_g. During the execution of the procedure, when $\beta_g \geq \beta$ is found, SEARCH_TOP_DOWN_AUX calls itself recursively in order to examine all

```
procedure SEARCH_TOP_DOWN(in (β_cur, p), (β, p); inout found, F; out (β_i, p, β'_i));
begin
    deeper:=false
    forall (β_k, p, β'_k) ∈ Sons(β_cur, p, β'_cur) do
        switch COMPARE(β_k, β) of
        case =:
            found:=true;
            (β_i, p, β'_i):=(β_k, p, β'_k);
            return;
        case >:
            deeper:=true
            SEARCH_TOP_DOWN((β_k, p), (β, p), found, F, (β_i, p, β'_i));
        case <:
            Fathers(β_k, p, β'_k):= Fathers(β_k, p, β'_k)\{(β_cur, p, β'_cur)};
        end;
    if not deeper then
        F:=F ∪ {(β_cur, p, β'_cur)}
end;

procedure SEARCH_BOTTOM_UP(in (β_cur, p), (β, p); inout S)
begin
    deeper:=false;
    forall (β_k, p, β'_k) ∈ Fathers(β_cur, p, β'_cur) do
        switch COMPARE(β_k, β) of
        case <:
            deeper:=true
            SEARCH_BOTTOM_UP((β_k, p), (β_p), S);
        case >:
            Sons(β_k, p, β'_k):= Sons(β_k, p, β'_k)\{(β_cur, p, β'_cur)};
        end;
    if not deeper then
        S:=S ∪ {(β_cur, p, β'_cur)}
end;
```

Figure 4.5: Original algorithm of SEARCH

sons of (β_g, p, β'_g). Since it must be in F, none of its sons have an input substitution greater or equal to β. Then the execution of the recursive call is over and, back to the examination of the sons of $(\beta, p, ?)$, since we have gone deeper, $(\beta, p, ?)$ will not be appended to F. It is in contradiction with the hypothesis. Hence, if the searched element is in F, there can not be any other element in F.

Let us now suppose that all $\beta_i (1 \leq i \leq n)$ are strictly greater than β. The fact that $(\beta_i, p, \beta'_i) \in F$ means the procedure has not gone deeper than that element, that is all sons of that element have an input substitution smaller or not comparable to β. So, there is no (α, p, α') in the Hasse diagram such that $\beta \leq \alpha \leq \beta_i$. Hence, every element of F is a father of $(\beta, p, ?)$. And, finally, all fathers of $(\beta, p, ?)$ are in F because the procedure stops its examination of the Hasse diagram when only elements with an input abstract substitution smaller or not comparable to β are still not examined, so all fathers have been checked. □

As a consequence, if F has more than one element, then we are sure β is not among them. But, if F has only one element, we do not know anything. It is then necessary to test the equality between the input substitution of that only element of F, and β.

For instance, let us search for $(\beta, p, ?)$ in the Hasse diagram shown on figure 4.3. If $\beta_5 < \beta < \beta_2$ then $F = \{(\beta_2, p, \beta'_2)\}$. If $\beta < \beta_5$, $\beta < \beta_6$ and $\bot < \beta$ then $F = \{(\beta_5, p, \beta'_5), (\beta_6, p, \beta'_6)\}$. And, if $\beta = \beta_5$, then $F = \{(\beta_5, p, \beta'_5)\}$.

Procedure `SEARCH_BOTTOM_DOWN` can be modified the same way.

Thus, we have now won one half of the tests, we know whether $(\beta, p, ?)$ is present in a Hasse diagram or not and we know both sets of its sons and its fathers. But the side effect is not performed any more. Then, in `SEARCH_BOTTOM_UP`, when the first test is not verified, the second test must be performed to determine whether $\beta_k > \beta$, in what case the side effect must be performed, or whether they ca not be compared. We still win on the number of tests, but no more the half. In `SEARCH_TOP_DOWN`, we ca not act the same way, since we do not know whether the elements appended to F are the searched one or not. Hence, we must do that in the end, in the case we are sure that the element is not in the diagram. We must examine all sons of every element of F a second time in order to perform the second test and remove the link between these elements of F and their sons that verify the test. The new procedures `SEARCH_TOP_DOWN` and `SEARCH_BOTTOM_UP` are shown on figure 4.6, procedure `SEARCH` is unchanged.

4.5.4 Comparison between both search methods

Let us now examine these two methods on both pathological cases shown on figure 4.7. Those are the two possible extremes in the shape of a Hasse diagram. In the first one, sat_h, all elements are in height, each element has one and only one father and one and only one son, but $\bot$ has a lot of ancestors and $\top$ has a lot of descendants. In the second one, sat_w, all elements are in width, every element different of $\top$ or $\bot$ is a son of $\top$ and a father of $\bot$. Let us suppose they both contain n elements.

```
procedure SEARCH_TOP_DOWN(in (β_cur, p), (β, p); inout found, F; out (β_i, p, β'_i));
begin
   SEARCH_TOP_DOWN_AUX((β_cur, p), (β, p), F);
   if #F = 1 then /* say F = {(β_F, p, β'_F)} */
      if SMALLER_OR_EQUAL(β, β_F) then begin
         found:=true;
         (β_i, p, β'_i):=(β_F, p, β'_F);
      end;
   if not found then
      forall (β_i, p, β'_i) ∈ F do
         forall (β_k, p, β'_k) ∈ Sons(β_i, p, β'_i) do
            if SMALLER_OR_EQUAL(β, β_k) then
               Fathers(β_k, p, β'_k):= Fathers(β_k, p, β'_k)\{(β_i, p, β'_i)};
end;

procedure SEARCH_TOP_DOWN_AUX(in (β_cur, p), (β, p); inout F);
begin
   deeper:=false;
   forall (β_k, p, β'_k) ∈ Sons(β_cur, p, β'_cur) do
      if SMALLER_OR_EQUAL(β, β_k) then begin
         deeper:=true;
         SEARCH_TOP_DOWN_AUX((β_k, p), (β_p, p), F);
      end;
   if not deeper then
      F:=F ∪ {(β_cur, p, β'_cur)};
end;

procedure SEARCH_BOTTOM_UP(in (β_cur, p), (β, p); inout S)
begin
   deeper:=false
   forall (β_k, p, β'_k) ∈ Fathers(β_cur, p, β'_cur) do
      if SMALLER_OR_EQUAL(β_k, β) then begin
         deeper:=true
         SEARCH_BOTTOM_UP((β_k, p), (β_p, p), S)
      end
      else if SMALLER_OR_EQUAL(β_k, β) then
         Sons(β_k, p, β'_k):= Sons(β_k, p, β'_k)\{(β_cur, p, β'_cur)};
   if not deeper then
      S:=S ∪ {(β_cur, p, β'_cur)};
end;
```

Figure 4.6: New algorithm of SEARCH

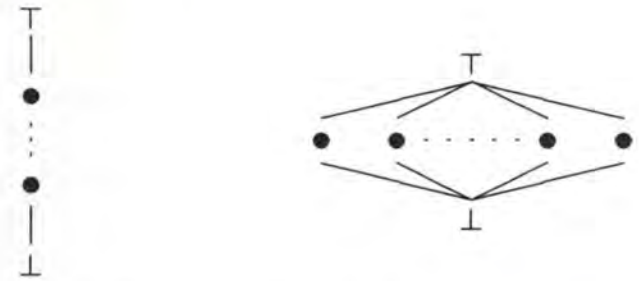

Figure 4.7: Two opposite pathological Hasse diagrams

What if the searched element is present in the diagram? With the first method, as we stop right when it is found, we shall look at $\frac{n}{2}$ elements from both diagrams in mean. Namely $\frac{n}{2} \times 2 = n$ tests "$\leq$". With the second method, $\frac{n}{2}$ elements of sat_h will be examined in mean with only one test "$\leq$". Then, one more test will be performed to check the only element in F. That is $\frac{n}{2} + 1$ tests. But, all the n elements of sat_w will be examined with the first test. Hence, $n + 1$ test are performed.

And what if the searched element is not in the diagram? With the original method, $\frac{n}{2}$ elements of sat_h are examined on the average, and all n elements of sat_w are examined. Moreover, this is done twice, once with `SEARCH_TOP_DOWN` and once with `SEARCH_BOTTOM_UP`. Hence, respectively $2n$ and $4n$ tests "$\leq$" are performed. With the new method, the number of tests is exactly the same sat_w, but exactly half of the tests are performed for sat_h, that is n and $4n$ tests "$\leq$" are computed respectively.

At the light of all these figures, we can draw conclusions. The new search method for an element in a Hasse diagram permits to reach our goal of dividing the number of tests by two, but only in some optimal pathological cases like sat_h, when the height of the diagram is maximum. And it can also make no difference, or even be worse than the original method in some other pathological cases, when the height is minimum (=1).

And in practice? In [15], statitics have been achieved about the shape of Hasse diagrams appearing during evaluation of the programs we tested in the previous chapters. All the average heights (including $\top$ and $\bot$) are ranging from 2.18 to 4.83 for an average number of elements (excluding appended $\top$ and $\bot$) ranging from 1.21 to 6.56. That is, the diagrams are more or less "well organized", namely their height is similar to their width. Moreover, these diagrams are not very big. So, we can hope to win one or two tests "$\leq$" during each search, but it's not very significant. Indeed, a few tests were achieved and no significant result appeared. On the other hand, reexecution that was experimented in the previous chapter, leads to the creation of more elements in the diagrams. So, with reexecution the new search may perhaps become more interesting. But no test has been done to check that assumption.

4.5.5 Implementation of Hasse diagrams

All elements of a Hasse diagram are related to the same predicate. Moreover, we always refer to a tuple in function of its predicate, but we never need to know to which predicate a tuple is associated. So, to each representation in memory of a predicate p, we associate a reference to its associated sat_p, but its not necessary to store a refernece to p in the representation of a tuple of sat_p. Hence, a Hasse diagram sat_p will be a set of pairs $(\beta_{in}, \beta_{out})$ instead of a set of tuples $(\beta_{in}, p, \beta_{out})$.

A *sat* always has at least two elements: $(\bot, \bot)$ and $(\top, ?)$. Furthermore, each element has fathers and sons, excepted $(\bot, \bot)$ that has no son and $(\top, ?)$ that has no father. Hence, a simple way of representing a *sat* is to have just two references: one to $(\bot, \bot)$ and one to $(\top, ?)$. All other elements will be linked to these ones by their relations father-son. As we do not know how many sons and how many fathers an element of a diagram can have, we will implement these relationships with linked lists of pointers. When a lattice is created, both pairs $(\bot, \bot)$ and $(\top, ?)$ are initialised and the first one is put in relation with the second one as its son.

As explained while we presented the dependency graph, each tuple must have a boolean value that tells whether that particular tuple must be reconsidered or not. By the way, to help to do statistics on the studied Prolog program, we can add a few other boolean values that can tell whether the predicate is recursive or not, locally recursive or mutually recursive,... It may be helpfull for reuse of the results by other programs such as compilers. That was done in [15]. But it is just complementary information and it is not necessary for the execution of the abstract interpretation algorithm, so we will not speak about these complementary informations any more.

Lastly, we will add a pointer for the creation of the sets of fathers and sons by procedure SEARCH. Thus, a set of sons is a linked list of elements of a *sat*. Let S be the set. S is a pointer to the first element of the set. That element has pointer to the second element and so on. To add an element to the list is easy, just copy the pointer S in the new element of the set and make S point to that new element. The same may be done for the set of fathers but, as an element of the set can not be at the same time the father and the son of a new element, one pointer is enough for both sets.

The declaration of the data types are shown on figure 4.8.

```
struct srelationship{
    node *parent;
    struct srelationship *next;
};
typedef struct srelationship trelationship;

struct snode {
    tas *betain;
    tas *betaout;
    short ToReconsider;
    trelationship *fathers;
    trelationship *sons;
    struct snode *set;
};
typedef struct snode tnode;

typedef struct {
    tnode *top;
    tnode *bottom;
};
```

Figure 4.8: Declarations for Hasse diagrams

originalité

Chapter 5

Clause prefix

The clause prefix technique is a method for optimizing the generic abstract algorithm by execution of only the clauses that need it and even of only the parts of those clauses that need it. Indeed, if we look at the execution of the original program on `APPEND/3`, figure 5.1, we can see that the same operation is done several times. For instance, the evaluation of the first clause is done three times, exactly the same. And, in the second clause for which the execution changes, both UNIF-FUN are computed the same way. It is obvious that it would be interesting to avoid those redundant calculations.

In this chapter, we will describe and implement a method to avoid these evaluations.

5.1 Theoretical background

5.1.1 Motivation

A predicate is reconsidered only if it is recursive (tail recursive, locally or mutually recursive). And, in the original program, every clause is reevaluated completely each time the predicate is considered. But, the non-recursive clauses of the procedure give the same result each time they are considered with the same input abstract substitution. And if a clause is recursive[1], all the goals that come before the first recursive procedure call or the first goal that calls recursively that predicate are unchanged. Hence, we aim to reconsider only the clauses that depend on an element that has been updated, and only from the goal that depends on the updated element.

5.1.2 Formalization

The key idea of this optimization is to modify the dependency graph. In the original program, the dependency graph just indicates that a predicate (say p_1) depends on

[1] A clause is said to be recursive if it belongs to a recursive procedure and it contains a call to that procedure, directly (locally recursive) or via a call to another procedure (mutually recursive).

```
TRY CLAUSE 1
  EXIT EXTC (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  CALL UNIF-FUN (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
  EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT CLAUSE 1
TRY CLAUSE 2
  EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
  EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
  EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
  CALL PRO-GOAL append(Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT PRO-GOAL append bottom
  EXIT EXTG bottom
  EXIT RESTRC bottom
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT CLAUSE 2
ADJUST
TRY CLAUSE 1
  EXIT EXTC (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  CALL UNIF-FUN (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
  EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT CLAUSE 1
TRY CLAUSE 2
  EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
  EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
  EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
  CALL PRO-GOAL append(Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT PRO-GOAL append(Gro(1):[],Gro(2),Gro(2))
  EXIT EXTG (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)),Gro(2),Gro(3):[],Gro(4))
  EXIT RESTRC (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)))
  EXIT UNION (Gro(1),Gro(2),Gro(3))
EXIT CLAUSE 2
ADJUST
TRY CLAUSE 1
  EXIT EXTC (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  CALL UNIF-FUN (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
  EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT CLAUSE 1
TRY CLAUSE 2
  EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
  EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
  EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
  CALL PRO-GOAL append(Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT PRO-GOAL append(Gro(1),Gro(2),Gro(3))
  EXIT EXTG (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Gro(3),Gro(6))
  EXIT RESTRC (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)))
  EXIT UNION (Gro(1),Gro(2),Gro(3))
EXIT CLAUSE 2
```

Figure 5.1: The Original Algorithm on append/3

another (say p_2). We will add to the graph which clauses with predicate p_1 depend on p_2 and for each of these clauses, the first goal that needs to be reconsidered. Since a built-in never calls the procedure it belongs to, the reconsideration of a clause will never begin at a built-in, only at a procedure call.

First, let us define what a clause prefix is.

Definition 15 Let c be a normalized clause and $g_1, \ldots, g_m$ the successive procedure calls in the body of c. Prefix i of clause c, say $c[i]$, is simply clause c truncated after procedure call $g_i (1 \leq i \leq m)$. The position (an integer) of the last goal of $c[i]$ in the clause c will be denoted by $last(c[i])$. To ease the presentation, we take the convention that prefix 0, say $c[0]$, will be the entire clause and $last(c[0])$ is the position of the first goal, may be a built-in, in c. Let us denote $nbproc(c[i])$ the number of procedure calls in the clause $c[i]$. Note that $last(c[i]), 1 \leq i \leq m$ corresponds to the length of $c[i]$ while $last(c[0])$ is always equal to 1.

Exemple:

Let $c(\ldots) : -b_1, b_2, g_1, b_3, g_2, b_4$ be a normalized clause. We will consider the following lines as prefixes of the clause c:

- $c[1] = b_1, b_2, g_1$
- $c[2] = b_1, b_2, g_1, b_3, g_2$
- $c[0] = b_1, b_2, g_1, b_3, g_2, b_4$

Next, let us modify the definition of the dependency graph to include clauses and clause prefixes.

Definition 16 A dependency graph dp is a set of tuples of the form $\langle (\beta, e), lt \rangle$ where e is a goal, a clause or a clause prefix and lt is a set $\{(\alpha_1, q_1), \ldots, (\alpha_n, q_n)\} (n \geq 0)$ such that, for each (β, e), there exists at most one lt such that $\langle (\beta, e), lt \rangle \in dp$.

During the execution of a clause, only the prefixes a goal of which they depend upon has been updated need to be reconsidered. The prefixes that can be avoided are in the dependency graph. Other ones are removed by REMOVE_DP, after ADJUST at the end of the previous consideration. The index of such a first prefix can be defined as

$$\texttt{FP}(\beta, c) = min\{i \mid (\beta, c[i]) \notin dom(dp) (0 \leq i \leq nbproc(c))\}$$

During the execution of a clause with substitution β, let us name β_{ext} the current substitution, namely the substitution which is the result of the previous goal and the argument for the current goal. We will use $logclause(\beta, c[i])$ to represent the value of β_{ext} before the execution of the goal $last(c[i]) (1 \leq i \leq nbproc(c))$.

All the definitions given in the previous chapter can now be updated or generalized to deal with clauses and clause prefixes and a few ones must be added.

- EXTC(c,β) is now used at the entry of a clause only if there is no prefix for which the calculations can be avoided.

- G_ADD_DP(**in** $\beta, p, c, i, \alpha, q$, **inout** dp) is a generalization of ADD_DP which takes into account clauses and clause prefixes. Informally speaking, this operation updates the dependency graph for a goal p, the clause c in which the goal appears, and all the relevant clause prefixes (i.e. those including the procedure call in position i).

  procedure G_ADD_DP(**in** $\beta, p, c, i, \alpha, q$, **inout** dp)
  begin
  ADD_DP(β, p, α, q, dp);
  ADD_DP(β, c, α, q, dp);
  for $k := i$ bf to $nbproc(c)$ **do**
  ADD_DP($\beta, c[k], \alpha, q, dp$)
  end

- G_EXT_DP(**in** β, p, **inout** dp) will include clauses and clause prefixes too.

  procedure G_EXT_DP(**in** β, p, **inout** dp)
  begin
  EXT_DP(β, p, dp);
  for $i := 1$ **to** m **with** $c_1, \ldots, c_m$ clauses of p **do**
  begin
  EXT_DP(β, c_i, p);
  for $j := 0$ **to** $nbproc(c_i)$ **do**
  EXT_DP($\beta, c[j], p$)
  end
  end

- G_REMOVE_DP(**in** $modified$, **inout** dp) is generalized in the sense that $modified$ is now a list of pairs (α_1, q_i) where q_i can either be a predicate, a clause or a procedure call.

- G_EXTEND(**in** β, p, **inout** sat) generalizes the operation EXTEND to initialize the prefixes.

  procedure G_EXTEND(**in** β, p, **inout** sat)
  begin
  EXTEND(β, p, sat);
  logclause($\beta, c[0]$) := EXTC(c, β)
  end

- FIRST_PREFIX$(\beta, c) = (last(c[\mathrm{FP}(\beta, c)]), logclause(\beta, c[\mathrm{FP}(\beta, c)]))$ is a function defined in order to simplify the algorithm. It returns the first subgoal of a clause to consider.
- MODIFIED_CLAUSES$(\beta, p) = \{c \mid (\beta, c) \notin dom(dp)$ and c is a clause of $p\}$ is the set of all clauses of procedure p to consider.

The new generic algorithm including clause prefixes is shown on figure 5.2

5.2 Implementation

The implementation is an upgrade of the original program. It is easily done with a few modifications of the data types (elements of the Hasse diagrams, the dependency graph) and some changes in a few procedures (ADD_DP, REMOVE_DP, SOLVE_CALL, SOLVE_GOAL, EXTEND). In this section, we shall explain those transformations and discuss a few choices we had to do in order to obtain the best possible result.

The implementation of the Hasse diagrams and the dependency graph in the original program are explained in the previous chapter. We shall now extend them in two phases. In the first one, we will add clauses. The clause prefixes will come in the second phase. In fact, the first phase is independent from the second one and can be implemented alone.

5.2.1 Clauses

To each predicate p/n of a Prolog program P is associated a *sat*: $sat_{p/n}$. $sat_{p/n}$ is a set of tuples $(\beta_{in}, p, \beta_{out})$, where no two β_{in} are equal and where β_{out} is an approximation of the result of (β_{in}, p). It is implemented as a Hasse diagram. To each element in the *sats* is associated a boolean value (*ToReconsider* that will be abreviated to *tr* in the following) which indicates whether the corresponding pair (β_{in}, p) must be reconsidered or not. Now, we must know that for each clause of the procedure. Further more, we do not reconsider each clause anymore, but we still need to know the result of each of them. So, those results must now be stored. Thus, we add to each element of *sat* a list of pairs made of a boolean value and a substitution. There is one pair for each clause of p indicating whether this particular clause is to be reconsidered or not and containing the resulting substitution of the clause. Each element of a *sat* is now of the form:

$$(\beta_{in}, p, \beta_{out}, tr, \{(b_1, \beta_1), \ldots, (b_n, \beta_n)\})$$

where n is the number of clauses with predicate p and (b_i, β_i) $(1 \leq i \leq n)$ are the pairs mentionned above.

Each element in the dependency graph (the one implemented, that is, the reversed one) originally indicated what (α_i, q_i) had to be reconsidered if the pair

```
procedure solve_call(in β_in, p, suspended; inout sat, dp)
begin
   if (β_in, p) ∉ (dom(dp) ∪ suspended) then
   begin
      if (β_in, p) ∉ dom(sat) then
         sat := G_EXTEND(β_in, p, sat);
      repeat
         β_out := ⊥;
         SC := MODIFIED_CLAUSES(β_in, p);
         G_EXT_DP(β_in, p, dp);
         for all c ∈ SC do
         begin
            solve_clause(β_in, p, c, suspended ∪ {(β_in, p)}, β_aux, sat, dp);
            β_out := UNION(β_out, β_aux)
         end;
         (sat, modified) := ADJUST(β_in, p, β_out, sat);
         REMOVE_DP(modified, dp)
      until (β_in, p) ∈ dom(dp)
   end
end

procedure solve_clause(in β_in, p, c, suspended; out β_out; inout sat, dp)
begin
   (f, β_ext) := FIRST_PREFIX(β_in, c);
   for i := f to m with b_1, ..., b_m body-of c do
   begin
      β_aux := RESTRG(b_i, β_ext);
      switch (b_i) of
      case X_j = X_k:
         β_int := AI_VAR(β_aux)
      case X_j = f(...):
         β_int := AI_FUNC(β_aux, f)
      case q(...):
         logclause(β_in, c[i]) := β_ext;
         solve_call(β_aux, q, suspended, sat, dp);
         β_int := sat(β_aux, q);
         if (β_in, p) ∈ dom(dp) then
            G_ADD_DP(β_in, p, c, i, β_aux, q, dp)
      end;
      β_ext := EXTG(b_i, β_ext, β_int)
   end;
   β_out := RESTRC(c, β_ext)
end
```

Figure 5.2: The Algorithm with the Clause Prefix Improvement

(β, p) to which it is associated is reconsidered. Now, the elements will contain some (α_i, q_i, c_i) where c_i is an integer identifying which clause of q_i to reconsider.

We can now update a few procedure to handle these new data.

- The procedure ADD_DP just takes one more parameter, namely c_i and stores it with the others in the dependency graph.
- The procedure REMOVE_DP, for each element of the dependency graph $\langle (\beta, p), lt \rangle$ for which the pair (β, p) has been reconsidered, sets the boolean value tr of each $(\alpha_i, q_i) \in lt$ to "to be reconsidered". It then destructs the graph. Now, for each $(\alpha_i, q_i, c_i) \in lt$, it will also mark the boolean value corresponding to the clause c_i (b_i) as "to be reconsidered".
- The procedure EXTEND is slightly modified in order to correctly initialize the new boolean values.
- The procedure SOLVE_CLAUSE is very slightly modified too. It just passes the indice of the clause to the procedure ADD_DP.
- The procedure SOLVE_CALL is the one that is the most difficult to change. In fact, we tried two ways of implementing it. Let us consider a procedure p and its clauses c_i. To execute the clause p with the input abstract substitution β is to execute every c_i with the β and to calculate the UNION of all resulting abstract substitutions. We do not reconsider every clause any more, but it is still necessary to calculate the UNION. That's why we need to store the result of each clause.

  A first way of doing this is still to look at every clause, each at its turn. If it must be reconsidered, it is done and the UNION is calculated between the result and the UNION of the previous clauses. Else, the result previously stored is recalled and used to calculate the UNION. Hence, by that way, it is still necessary to calculate a great number of UNION operations.

  The second way makes it possible to reduce the number of UNION operations to calculate. It is based on the fact that the UNION operation, in the domains implemented, is "accumulative", namely the result of the operation is greater then its arguments, and that the algorithms converges in a growing way toward an element of $\mu(TSAT)$. It suffices to compute the UNION between the results of the reconsidered clauses and the result of the previous consideration of (β, p). Its main advantage is that it is not necessary to store the results of every clause any more. The *sat*'s are then of the following form:

$$(\beta_{in}, p, \beta_{out}, tr, \{b_1, \ldots, b_n\})$$

  Even if the second method is simpler, all the following has been done with the first implementation but time measurements have been done with the second one too and the precision of the clock is too weak to see a difference.

5.2.2 Clause prefixes

In a second time, we will implement the notion of prefix and use it to reduce the number of computations in the evaluation of a clause.

As explained in the previous section, we will only consider the prefixes that end with a procedure call. During the first consideration of a clause, it is not possible to know which prefixes could be avoided in the following reconsiderations. And, for the first procedure call to reconsider, it is necessary to know what the substitution calculated so far is. Thus, it is necessary to store the current abstract substitution before each $last(c[i])$.

Furthermore, when a procedure call arises, three steps are performed: the restriction of the current substitution to the arguments of the call, the call in itself and the extension of the result to the variables of the clause. If a procedure call is the first considered of a clause, i.e. all the previous prefixes have been avoided, then the first step will be exactly the same as the previous time it was done. It can be avoided too, simply by storing its result. So, for each prefix that begins with a procedure call, two substitutions are saved. It is necessary to keep the first one even if the second is sufficient for the call in order to compute the extension in the third step.

Elements of *sat*'s will now be of the form:

$$(\beta_{in}, p, \beta_{out}, tr, \{(b_1, g_1, \{(\beta_1^1, \beta_1'^1), \ldots, (\beta_1^{n_1}, \beta_1'^{n_1})\}), \ldots,$$

$$(b_n, g_n, \{(\beta_n^1, \beta_n'^1), \ldots, (\beta_n^{n_n}, \beta_n'^{n_n})\})\})$$

where n_i is the number of procedure calls in clause i, g_i $(1 \leq i \leq n)$ $(1 \leq g_i \leq n_i)$ means that clause i must be reconsidered from prefix $c[g_i]$, β_i^j $(1 \leq i \leq n$ and $1 \leq j \leq n_i)$ is the calculated substitution before the subgoal g_i and $\beta_i'^j$ $(1 \leq i \leq n$ and $1 \leq j \leq n_i)$ is the restricted substitution input of the subgoal g_i.

The implementation of the new data type for those structures is shown on figure 5.3. It is the ones presented in the previous chapter that are completed with linked lists to store informations for each clause and for each clause prefix.

The dependency graph can now be completed to handle prefixes. An element of the set lt of an element of that graph is now of the form

$$(\alpha_i, q_i, c_i, g_i)$$

where α_i is the input substitution of the predicate q_i, c_i is the particular clause to reconsider if (α_i, q_i) is reconsidered and g_i is the subgoal of c_i to be reconsidered. When a pair (β, p) is reexamined, all the elements of its dependency graph are to be reconsidered too, i.e. the tr boolean value associated to $(\beta, p, sat(\beta, p))$ must be set to *true*, the boolean value b_i associated to each clause c_i that must be reconsidered is set to *true* too and, g_i the indice of the first goal of the clause to reexamine must be set to the minimum of its current value and of the value stored in the element of the dependency graph.

```
struct srelationship {
    node *parent;
    struct srelationship *next;
};
typedef struct srelationship trelationship;

struct sgoal {
    tas *betain;
    tas *betarestr;
    struct sgoal *next;
};
typedef struct sgoal tgoal;

struct sclause {
    short ToReconsider;
    int firstgoal;
    tgoal *goals;
    struct sclause *next;
};
typedef struct sclause tclause;

struct snode {
    tas *betain;
    tas *betaout;
    short ToReconsider;
    tclause *clause;
    trelationship *fathers;
    trelationship *sons;
    struct snode *set;
};
typedef struct snode tnode;

typedef struct {
    tnode *top;
    tnode *bottom;
};
```

Figure 5.3: Declarations for Hasse diagrams

For instance, let us assume the following is an element of the dependency graph:

$$\langle(\beta, p), \{\ldots, (\alpha, q, c, g), \ldots\}\rangle$$

and let us assume the following is an element of sat_q:

$$(\alpha_{in}, q, \alpha_{out}, tr, \{\ldots, (b, g', \{(\beta^1, \beta'^1), \ldots, (\beta^n, \beta'^n)\}), \ldots\})$$

Then, when an examination of (β, p) is over, that element of sat_q will become:

$$(\alpha_{in}, q, \alpha_{out}, true, \{\ldots, (true, min(g, g'), \{(\beta^1, \beta'^1), \ldots, (\beta^n, \beta'^n)\}), \ldots\})$$

Procedures ADD_DP and REMOVE_DP can be updated easily in order to handle new data in the way described above. EXTEND must be modified to initialize correctly those new data.

Procedure SOLVE_CLAUSE is changed too with the adjunction of an initialisation of the indice of first subgoal to consider and of the current substitution just before that subgoal with values stored in the *sat*.

5.3 The result: Append

Figure 5.4 displays the execution on the example we already looked at: `APPEND/3`. A simple comparison with the execution of the original program on the same test, as shown on figure 5.1, shows the obvious interest of this optimization.

```
TRY CLAUSE 1
  EXIT EXTC (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  CALL UNIF-FUN (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
  CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
  EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
  EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT CLAUSE 1
TRY CLAUSE 2
  EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
  CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
  EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
  CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
  EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
  CALL PRO-GOAL append (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT PRO-GOAL append bottom
  EXIT EXTG bottom
  EXIT RESTRC bottom
  EXIT UNION (Gro(1):[],Gro(2),Gro(2))
EXIT CLAUSE 2
ADJUST
TRY CLAUSE 2
  EXIT PREFIX (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
  CALL PRO-GOAL append (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT PRO-GOAL append (Gro(1):[],Gro(2),Gro(2))
  EXIT EXTG (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)),Gro(2),Gro(3):[],Gro(4))
  EXIT RESTRC (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)))
  EXIT UNION (Gro(1),Gro(2),Gro(3))
EXIT CLAUSE 2
ADJUST
TRY CLAUSE 2
  EXIT PREFIX (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
  CALL PRO-GOAL append (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
  EXIT PRO-GOAL append (Gro(1),Gro(2),Gro(3))
  EXIT EXTG (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Gro(3),Gro(6))
  EXIT RESTRC (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)))
  EXIT UNION (Gro(1),Gro(2),Gro(3))
EXIT CLAUSE 2
```

Figure 5.4: The Clause Prefix Algorithm on append/3

Chapter 6

Caching

6.1 Introduction

During execution of `SolveGoal`, a lot of abstract operations are executed. Many of them are called with arguments already encountered. Obviously the result is the same if there is no side effect. The idea of the *Caching* optimization is to memoize all abstract operations. So when the original algorithm computes a clause C, if C wasn't considered by the *Prefix* version then all asbtract operations which make up C have already been computed. So it's sufficient to find the result of each operation (previously stored in a table). Furthermore we can trap some operations in *Caching* normally computed in *Prefix* version.

Thus we should expect that *Caching* will be as good as *Prefix*. Unfortunately the memoization implies to maintain a table of all computed results and to look up this table to cache an operation already encountered. Thus the memoization causes a additional consumption of CPU time in comparison with *Prefix*. When *Prefix* avoids simply an operation, *Caching* must (a) call a function, (b) detect that the result was already computed , and (c) return it.

Figure 6.1 depicts the execution of the `append/3` program. As it can be noticed, all operations on the first clause as well as all operations up to the recursive call in the second clause are cached and therefore automatically reused by the algorithm. In this particular case, no further improvement is brought by the caching improvement. However, in other programs, other results will be shared.

6.2 The Implementation

6.2.1 Memoization

For each abstract operation ω we have a set S_ω. If ω is defined as:

$$\omega : A_1 \times A_2 \times \ldots \times A_n \longrightarrow B$$

```
                TRY CLAUSE 1
                    EXIT EXTC (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
                    CALL UNIF-FUN (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
                    EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
                    CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
                    EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
                    EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
                    EXIT UNION (Gro(1):[],Gro(2),Gro(2))
                EXIT CLAUSE 1
                TRY CLAUSE 2
                    EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
                    CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
                    EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
                    CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
                    EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
                    CALL PRO-GOAL append(Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
                    EXIT PRO-GOAL append bottom
                    EXIT EXTG bottom
                    EXIT RESTRC bottom
                    EXIT UNION (Gro(1):[],Gro(2),Gro(2)) k ps:
                EXIT CLAUSE 2
                ADJUST
                TRY CLAUSE 1
CACHED              EXIT EXTC (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
                    CALL UNIF-FUN (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
CACHED              EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
                    CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
CACHED              EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
CACHED              EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
CACHED              EXIT UNION (Gro(1):[],Gro(2),Gro(2))
                EXIT CLAUSE 1
                TRY CLAUSE 2
CACHED              EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
                    CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
CACHED              EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
                    CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
CACHED              EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
CACHED              RESTRG CACHED
                    CALL PRO-GOAL append(Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
                    EXIT PRO-GOAL append(Gro(1):[],Gro(2),Gro(2))
                    EXIT EXTG (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)),Gro(2),Gro(3):[],Gro(4))
                    EXIT RESTRC (Gro(1):.(Gro(2),Gro(3):[]),Gro(4),Gro(5):.(Gro(2),Gro(4)))
                    EXIT UNION (Gro(1),Gro(2),Gro(3))
                EXIT CLAUSE 2
                ADJUST
                TRY CLAUSE 1
CACHED              EXIT EXTC (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
                    CALL UNIF-FUN (Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
CACHED              EXIT UNIF-FUN (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
                    CALL UNIF-VAR (Gro(1):[],Var(2),Gro(3)) ps: (2,2)
CACHED              EXIT UNIF-VAR (Gro(1):[],Gro(2),Gro(2))
CACHED              EXIT RESTRC (Gro(1):[],Gro(2),Gro(2))
CACHED              EXIT UNION (Gro(1):[],Gro(2),Gro(2))
                EXIT CLAUSE 1
                TRY CLAUSE 2
CACHED              EXIT EXTC (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
                    CALL UNIF-FUN (Var(1),Var(2),Gro(3),Var(4),Var(5),Var(6)) ps: (1,1)(2,2)(4,4)(5,5)(6,6)
CACHED              EXIT UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
                    CALL UNIF-FUN (Ngv(1):.(Var(2),Var(3)),Var(4),Gro(5),Var(2),Var(3),Var(6)) ps: (2,2)(3,3)(4,4)(6,6)
CACHED              EXIT UNIF-FUN (Ngv(1):.(Gro(2),Var(3)),Var(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Var(3),Gro(6)) ps: (3,3)(4,4)
CACHED              RESTRG CACHED
                    CALL PRO-GOAL append(Var(1),Var(2),Gro(3)) ps: (1,1)(2,2)
                    EXIT PRO-GOAL append(Gro(1),Gro(2),Gro(3))
                    EXIT EXTG (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)),Gro(2),Gro(3),Gro(6))
                    EXIT RESTRC (Gro(1):.(Gro(2),Gro(3)),Gro(4),Gro(5):.(Gro(2),Gro(6)))
                    EXIT UNION (Gro(1),Gro(2),Gro(3))
                EXIT CLAUSE 2
```

Figure 6.1: The Caching Algorithm on append/3

then S_ω is defined as part of

$$(A_1 \times \ldots \times A_n) \times B$$

The implementation of each operator ω is replaced by:

let $(a_1, \ldots, a_n)$ be the argument
if $\exists\ X \in B:\ \ ((a_1, \ldots, a_n), X) \in S_\omega$
then result $\leftarrow X$
else $y := \omega(a_1, \ldots, a_n)$
$S_\omega := S_\omega \cup \{((a_1, \ldots, a_n), y)\}$
result $:= y$

This definition ensures us that operator ω is evaluated only once for each argument.

The set S_ω is implemented with a hash table. All the elements that hash to the same slot are put in a linked list. To build the chain, we take cells from big overflow tables allocated at the beginning of execution. This allows us to save time and memory by avoiding to break the memory space into little segments. When some arguments are substitutions, it may be very expensive to apply an equality test when looking for an element in tables. The solution is explained in the next chapter. The data type is the same for all sets even though arguments and results are different from one operator to another. Following definitions are necessary to understand how this is possible.

Let Ω be the set of all operators that we wish to memoize. Let us define

$$maxarity = \max_{\omega \in \Omega} arity(\omega)$$

$$cardo = \sharp\Omega$$

if the arity of ω_i is smaller than $maxarity$, we extend its domain with a set $STUFF$ as many times as it is necessary to obtain an arity $maxarity$. $STUFF$ is a set without importance that you can fix to N for instance. So each operator is defined now as:

$$\omega : A_1^\omega \times \ldots \times A_{maxarity}^\omega \longrightarrow B^\omega$$

We define

$$Do_i = A_i^{\omega_1} + \ldots + A_i^{\omega_{cardo}} \quad \forall i \quad 1 \leq i \leq cardo$$

$$Co = B^{\omega_1} + \ldots + B^{\omega_{cardo}}$$

We have now a generic operator γ defined as:

$$\gamma : Do_1 \times \ldots \times Do_{maxarity} \longrightarrow Co$$

So if a memoization process exists for γ, we can systematically apply it to other operators. The disjoint union $+$ is defined as union $\cup$ except that if $x \in A + B$ it is always possible to know if $x \in A$ or $x \in B$. The disjoint union $+$ may easily be implemented with a typecasting (or a *union* data structure) in the C language. Thus the sets S_ω may have the same definition for each operator. It allows us to use the same code for every operator without loss of time.

6.2.2 Abstract Substitutions

In the previous implementation, the comparison of two substitutions implies an exam of the content. This one requires an heavy computation which can be avoided if substitutions have an unique representation in memory. This property insures us that if x and y are two C-variables denoting substitutions, then we have the following property (using the C language notation, $\&x$ denotes the address where the content of x is stored)

$$x == y \Rightarrow \&x == \&y$$

So we avoid a time consuming equality test at the time of looking for the presence of an argument in S_ω. We compare only two pointers instead of two substitutions. We introduce here a definition:

Definition 17 [Direct Substitution] a direct substitution is a pointer to a substitution that may be modified at any time by its creator only. Let D be the set of all possible direct substitutions.

Let $I \subset N$ be the set of all identifiers which are available. Let S be the set of all substitutions used at a time. S is defined as

$$S \subset D \times N \times N$$

where the first component is a pointer to a substitution, the second is a reference counter and the third an identifier (if $s \in S$ and $s = (a, b, i)$ then i denotes the real address of s in memory).

When a procedure has to build a new substitution, it uses local memory to construct the substitution, then invokes the procedure *NewSubst* in order to update the set S. *NewSubst* returns an identifier referring to the substitution. The procedure may destroy the substitution in local memory and continue his work with the identifier. When a procedure has to destroy a substitution belonging to S, it calls *RemoveSubst* in order to update S. In no way a procedure can modify directly a substitution belonging to S.

When the program starts, S and I are initialized to $\emptyset$ and N. NewSubst0 and RemoveSubst0 are respectively the constructor and destructor for the abstract substitutions. Their definition are:

The Constructor NewSubst0: $D \longrightarrow D \times N \times N$

pre $S = S_0$
$I = I_0$
let d be the argument
post if $\exists (e, r, i) \in S$: e and d represent the same substitution

then $S = S_0\langle(d, r, i) \rightarrow (d, r+1, i)\rangle$
else $I = I_0 - \{j\}$ where $j \in I_0$
$S = S_0 \cup \{(e, 1, j)\}$ where $e = copy(d)$ [:]

[:] $Copy(\bullet)$ returns a pointer to a copy of its argument. This operation consumes CPU time but it is necessary to maintain the integrity of the set S.

The destructor RemoveSubst0: $D \times N \times N \longrightarrow \emptyset$

pre let (d, r, i) the argument
$S = S_0$
$(d, r, i) \in S_0$ and $r > 0$
post $S = S_0\langle(d, r, i) \rightarrow (d, r-1, i)\rangle$

The set S is implemented with a hash table. Its organization is the same as for memoization tables. When a reference counter becomes null, the substitution associated to this entry should be destroyed and the cell should be extracted from the linked list. This policy impairs the performances of memoization. So cells with a null reference counter are let in the list and the substitution is not destroyed. Consequently unused substitutions may cause some problems when the available memory becomes critical. When it occurs it is necessary to release all the cells (and their substitutions) with a null reference counter.

6.2.3 Memory Cleaning

It's obvious that keeping unreferenced substitutions alive in memory is mandatory. This ensures us these two conditions:

- a substitution result (an identifier) returned by memoization is always valid
- when an identifier matches with an other identifier stored among the arguments of a memoization table, these two identifiers represent the same substitution.

Now, this may cause an memory overflow which would not occur if unused substitutions were destroyed. A call to a cleaning procedure when we detect a lack of memory solves this problem. But it's necessary to modify the memory management. We redefine S as

$$S \subset D \times N \times N \times N$$

where the fourth component is a magic number and the rest as before. The magic number helps us to garantee the integrity of the set S. When a substitution in S is modified, its magic number is incremented. So all references to the old substitution can be detected and managed in consequence to avoid a conflict.

The constructor is redefined as: NewSubst1 : $D \longrightarrow D \times N \times N \times N$

pre let d the argument
$S = S_0$
$I = I_0$
post if $\exists(e,r,i,m) \in S_0$: e and d represent the same substitution
then $S = S_0\langle(e,r,i,m) \rightarrow (e,r+1,i,m)\rangle$
result= $(e,r+1,i,m)$
else $\exists(f,s,j,m) \in I_0$
$I = I_0 - \{(f,s,j,m)\}$
$S = S_0 \cup \{(d',1,j,m+1)\}$ where $d' = copy(d)$
result= $(d',1,j,m+1)$

I is here a set which collects all unused tuples (and their identifiers).
The destructor is redefined as: Removesubst1: $D \longrightarrow D \times N \times N \times N$

pre let (d,r,i,m) the argument
$S = S_0$
$I = I_0$
$(d,r,i,m) \in S_0 \ \wedge \ r > 0$
post $S = S_0\langle(d,r,i,m) \rightarrow (d,r-1,i,m)\rangle$

The procedure *Clean* to remove all unused substitutions is defined as:

pre $S = S_0$
$I = I_0$
$(d,r,i,m) \in I \Rightarrow r = 0$
post $(d,r,i,m) \in S \Leftrightarrow ((d,r,i,m) \in S_0) \wedge (r \neq 0)$
$(d,r,i,m) \in I \Leftrightarrow ((d,r,i,m) \in S_0 \cup I_0) \wedge (r = 0)$

The procedure *Clean* examines each entry of the hash table (and the associated linked list). Each cell whose reference counter is null, is removed from the hash table and the substitution is physically destroyed to release its memory space. It's very important to increase its magic number to prevent memoization to reuse a bad substitution. The memoize process must manage explicitly this magic number to verify if an identifier is valid or not. Now, a substitution is identified by its identifier (third component) inside a hash table and the magic number is needed to identify the substitution while the program runs.

The memoization process has to be rethought:

Let's define

$$\omega : A_1 \times \ldots \times A_n \longrightarrow B$$

Then S_ω is defined as

$$(A_1 \times \ldots \times A_n) \times B \times (\underbrace{N \times \ldots \times N}_{m \text{ elements}})$$

where m (resp. p) is the number of sets in $(A_1, \ldots, A_n, B)$ (resp. $(A_1, \ldots, A_n)$) representing substitutions. Let $(i_1, \ldots, i_p)$ be the indexes of substitutions in $(A_1, \ldots, A_n)$. Let $(a_1, \ldots, a_n)$ be the argument. Memoization is implemented as depicted in figure 6.2

6.3 Performances

We shall examine each algorithm to discuss its performance and its implementation.

6.3.1 The Memory Manager For The Substitutions

The time consuming operations are the hash function and the search of a substitution in the hash table. Since the substitutions are represented in a unique way, the hash function may be computed directly on the bytes composing a substitution. If $(b_i)_1^n$ are the n bytes composing the substitution θ then

$$\text{hash}(\theta)=\text{hash}((b_i)_1^n)=\begin{array}{ll} b_n \text{ xor } 3 \times hash((b_i)_1^{n-1}) & \text{if } n \geq 1 \\ 0 & \text{otherwise} \end{array}$$

The hash function is domain-specific and uses only worthwhile fields (mode, same value and pattern for the *type* domain). There is less than 30% of collisions. It is extremely fast to test if an entry in the hash table is equal to a substitution since two substitutions have the same representation.

6.3.2 The Memoization Process

The search in a memoization table is extremely fast. The hash function is the same as for substitutions. To test the equality of two entries it's sufficient to test some integers. Substitutions (identifiers/pointers) returned by memoization are always valid except if some garbage collection is done (procedure *clean*).

procedure(in $a_1, \ldots, a_n$,**out** X,**in** $Y_1, \ldots, Y_n$)
begin
  if $\exists((a_1, \ldots, a_n), X, (Y_1, \ldots, Y_m)) \in S_\omega$ where X, Y_i are unknown
  then begin
    if $(magic(S, a_{j_i}) = Y_i \ \ \forall i \in \{1, \ldots, p\})$
      $\wedge(X$ is a substitution $\Rightarrow magic(S, X) = Y_m)$
    then $R := X$
    else begin
      $R := \omega(a_1, \ldots, a_n)$
      $Z_i := magic(S, a_{j_i})\ \forall i \in \{1, \ldots, p\})$
      if R is a substitution
      then $Z_m := magic(S, R)$
      $S_\omega := S_\omega \langle((a_1, \ldots, a_n), X, (Y_1, \ldots, Y_m)) \rightarrow ((a_1, \ldots, a_n), R, (Z_1, \ldots, Z_m))\rangle$
    end
  end
  else begin
    $R := \omega(a_1, \ldots, a_n)$
    $Z_i := magic(S, a_{j_i})\ \forall i \in \{1, \ldots, p\})$
    if R is a substitution
    then $Z_m := magic(S, R)$
    $S_\omega := S_\omega \cup \{((a_1, \ldots, a_n), X, (Y_1, \ldots, Y_m)) \rightarrow ((a_1, \ldots, a_n), R, (Z_1, \ldots, Z_m))\}$
  end
  result $:= R$
end

$magic(S, x)$ represents the magic number associated with x in the set S.The memoization process is, as previously explained, generic for all cached operations.

Figure 6.2: Memoization Procedure

Chapter 7

Experimental Evaluation

In this section, we report our experimental results on the optimization techniques. In the following, we denote respectively by `Pascal`, `Original`, `Prefix` and `Caching` (`Pa`,`Or`,`Pr`,`Ca` for short) the original algorithm coded in Pascal, the original algorithm coded in C with a number of optimization techniques on the domain implementation, the algorithm with the clause prefix improvement, and the algorithm with the caching improvement. The optimization techniques of `Original` over `Pascal` include a lazy computation of the transitive closure of the sharing component (i.e. call by need) and a data-driven implementation (instead of a straightforward top-down implementation) of various operations on substitutions (in particular the unification operation).

Section 7.1 describes the programs used in the experiments. Section 7.2 describes the computation times of the algorithms. Section 7.3 describes the number of operations on substitutions performed by each of the algorithms and the hit-ratios of the caches. Section 7.4 depicts the time distribution between control and abstract operations as well as the time distribution among the various abstract operations, while Section 7.5 reports the memory consumption of the algorithms. Finally, Section 7.6 gives the results on a simple abstract domain.

7.1 The Programs

The programs we use are hopefully representative of "pure" logic programs (i.e. without the use of dynamic predicates such as `assert` and `retract`). They are taken from a number of authors and used for various purposes, from compiler writing to equation-solvers, combinatorial problems, and theorem-proving. Hence they should be representative of a large class of programs. In order to accommodate the many built-ins provided in Prolog implementations and not supported in our current implementation, some programs have been extended with some clauses achieving the effect of the built-ins. Examples are the predicates to achieve input/output, meta-predicates such as `setof`, `bagof`, `arg`, and `functor`. The clauses containing `assert` and `retract` have been dropped in the one program containing them (i.e. Syntax

error handling in the reader program).

The program `kalah` is a program which plays the game of kalah. It is taken from [22] and implements an alpha-beta search procedure. The program `press` is an equation-solver program taken from [22] as well. We use two versions of this interesting program. The first version is the standard version (`press1`) while the second version (`press2`) has a goal repeated in the program (i.e. a goal is executed twice in a clause). The two versions illustrate a fact often neglected in abstract interpretation. A more precise domain, although requiring a higher cost for the basic operations, might in fact be much more efficient since fewer elements in the domain are explored. The repetition of some goals in the `Press` program allows us to simulate a more precise domain (and hence to gain efficiency). The program `cs` is a cutting-stock program taken from [23]. It is a program used to generate a number of configurations representing various ways of cutting a wood board into small shelves. The program uses, in various ways, the nondeterminism of Prolog. The program `Disj` is taken from [9] and is the generate and test equivalent of a constraint program used to solve a disjunctive scheduling problem. This is also a program using the nondeterminism of Prolog. The program `Read` is the tokeniser and reader written by R. O'keefe and D.H.D. Warren for Prolog. It is mainly a deterministic program, with mutually recursive procedures. The program `PG` is a program written by W. Older to solve a specific mathematical problem. The program `Gabriel` is the `Browse` program taken from Gabriel benchmark. The program `Plan` (PL for short) is a planning program taken from Sterling & Shapiro. The program `Queens` is a simple program to solve the n-queens problem. `Peep` is a program written by S.Debray to carry out the *peephole* optimization in the SB-Prolog compiler. It is a deterministic program. We also use the traditional concatenation and quicksort programs, say `Append` and `Qsort` (difference lists).

7.2 Computation Times

We give two versions of the computation times. Table 7.1 depicts the results with the sharing represented by characters (i.e. bytes) while Table 7.2 depicts the results with the sharing represented by bits. The first four columns present the computation times in seconds while the last five columns present the improvement in percentage (i.e. `P1-P2` denotes `(P1-P2)/P1`).

As far as the character version is concerned, `Caching` produces an improvement of 58.42% compared to the original version in Pascal. `Caching` also produces an improvement of 28.31% compared to the original version in C. Programs `Read` and `Peep` are those producing the least improvement (44.53% and 49.67%) while `Disj` and `Kalah` produce the best improvement (73.49% and 68.63%). All the times are below 10 seconds except `Press1` and `Read` which require respectively 25.62 and 25.37 seconds. `Prefix` is marginally faster than `Caching`. It produces an average improvement of 58.49% over the original implementation and 28.38% over the im-

Program	Pa	Or	Pr	Ca	Pa-Or	Pa-Pr	Or-Pr	Pa-Ca	Or-Ca
Append	0.06	0.02	0.01	0.01					
Kalah	17.82	9.20	6.33	5.59	48.37	64.48	31.20	68.63	39.24
Queens	0.37	0.22	0.15	0.16	40.54	59.46	31.82	56.76	27.27
Press1	65.91	37.89	28.82	25.62	42.51	56.27	23.94	61.13	32.38
Press2	19.52	11.53	8.65	8.36	40.93	55.69	24.98	57.17	27.49
Peep	11.34	7.14	5.79	6.29	37.04	48.94	18.91	44.53	11.90
CS	16.02	7.93	5.70	5.92	50.50	64.42	28.12	63.05	25.35
Disj	12.26	6.75	3.03	3.25	44.94	75.29	55.11	73.49	51.85
PG	1.79	1.11	0.86	0.76	37.99	51.96	22.52	57.54	31.53
Read	50.41	30.26	24.42	25.37	39.97	51.56	19.30	49.67	16.16
Gabriel	4.88	2.89	1.95	2.06	40.78	60.04	32.53	57.79	28.72
Plan	1.26	0.71	0.59	0.60	43.65	53.17	16.90	52.38	15.49
QSort	0.56	0.34	0.22	0.23	39.29	60.71	35.29	58.93	32.35
Mean					42.21	58.49	28.38	58.42	28.31

Table 7.1: Computation Times of the Algorithms and Percentages: Character Version

Program	Pa	Or	Pr	Ca	%Pa-Or	%Pa-Pr	%Or-Pr	%Pa-Ca	%Or-Ca
Append	0.06	0.03	0.02	0.02					
Kalah	17.82	13.52	9.30	7.95	24.13	47.81	31.21	55.39	41.20
Queens	0.37	0.30	0.18	0.18	18.92	51.35	40.00	51.35	40.00
Press1	65.91	53.03	40.52	34.68	19.54	38.52	23.59	47.38	34.60
Press2	19.52	16.06	12.23	11.32	17.73	37.35	23.85	42.01	29.51
Peep	11.34	9.98	8.08	8.62	11.99	28.75	19.04	23.99	13.63
CS	16.02	11.67	8.49	8.43	27.15	47.00	27.25	47.38	27.76
Disj	12.26	9.97	4.49	4.64	18.68	63.38	54.96	62.15	53.46
PG	1.79	1.53	1.19	1.09	14.53	33.52	22.22	39.11	28.76
Read	50.41	43.36	35.51	35.82	13.99	29.56	18.10	28.94	17.39
Gabriel	4.88	4.13	2.74	2.73	15.37	43.85	33.66	44.06	33.90
Plan	1.26	0.99	0.80	0.78	21.43	36.51	19.19	38.10	21.21
QSort	0.56	0.47	0.32	0.28	16.07	42.86	31.91	50.00	40.43
Mean					18.29	41.70	28.75	44.15	31.82

Table 7.2: Computation Times of the Algorithms and Percentages: Bit Version

Program	Original	Prefix	Caching
Kalah	31.95	31.94	29.69
Queens	26.67	16.67	11.11
Press1	28.55	28.87	26.12
Press2	28.21	29.27	26.15
Peep	28.46	28.34	27.03
CS	32.05	32.86	29.77
Disj	32.30	32.52	29.96
PG	27.45	27.73	30.28
Read	30.21	31.23	29.17
Gabriel	30.02	28.83	24.54
Plan	28.28	26.25	23.08
QSort	27.66	31.25	17.86
Mean	29.32	28.81	25.40

Table 7.3: Percentage Gained By Using Characters on the Algorithms

proved implementation in C. All the programs are still under 28 seconds and **Prefix** loses around 3 seconds on one of the big programs.

As far as the bit version is concerned, **Caching** produces an improvement of 44.15% over the Pascal implementation (Booleans are not coded as bits by the Pascal compiler) and 31.82% over the improved C implementation. All programs still run below 12 seconds except **Press1** and **Read** which take respectively 34.68 and 35.82 seconds. **Prefix** is slower with an average improvement of 41.70% over the Pascal implementation and an average of 28.75% over the improved C implementation.

The results seem to indicate that the more costly the abstract operations, the more attractive **caching** will be. On our domain, the character implementation of sharing (which is the fastest) produces a gain of 0.07 % in favor of **Prefix** while the bit implementation produces a gain of 2.45 % in favor of caching. We discuss this result later in the paper in light of other results.

The above results compare well with the specialized algorithms of [25, 11]. On **Peep**, **Read** and **PG**, their best programs achieve respectively 22.52, 60.18 and 3.25 on a SUN 3/50. This means that our algorithm is respectively 3.89, 2.46, 4.27 times faster on a SPARC-I (which is around 2-4 times faster). Moreover, our algorithms are executed on a more sophisticated and accurate domain than the one used in [25, 11]. In particular, our domain also includes sharing and pattern information omitted in [25, 11].

Table 7.3 indicates the gain of using characters instead of bits on **Original**, **Prefix** and **Caching** for the sharing components. The improvement obtained is fairly consistent among the algorithms and is in general about 26-27%.

In short, the two improvements produce substantial gain in efficiency. Even after a gain of around 40% obtained by the C implementation by refining the abstract domain algorithms, they still produce an improvement of around 30%. Depending upon the implementation of the sharing component (to favor memory or speed),

Operation	Or	Pr	Ca	Ca eval	% Or-Pr	% Or-Ca	% Or-Ca eval
COMPARE	7294	5994	3493	1736	17.82	52.11	76.20
SMALLER	24840	20390	13329	8428	17.91	46.34	66.07
EXTEND	3416	2370	3416	987	30.62	0.00	71.11
AI_TEST	934	565	934	462	39.51	0.00	50.54
AI_IS	513	303	513	240	40.94	0.00	53.22
AI_VAR	896	615	896	566	31.36	0.00	36.83
AI_FUNC	13916	9086	13916	8208	34.71	0.00	41.06
EXTG	4982	3879	4982	3334	22.14	0.00	33.08
RESTRG	4982	2942	4982	2442	40.95	0.00	50.98
EXTC	5170	3388	5170	3388	34.47	0.00	34.47
RESTRC	5170	4325	5170	2704	16.34	0.00	47.70
UNION	9068	8468	9068	5349	6.62	0.00	41.01

Table 7.4: Number of Abstract Operations on all Programs for all Algorithms

Caching is slower (character version) or faster (bit version) than Prefix. Finally, the algorithm efficiency is at least as good as the best specialized tools available for these tasks, although it uses a more sophisticated domain and provides more accurate results (see [15] for details).

7.3 Number of Abstract Operations

To avoid considering the specificities of our implementation, we now give a more abstract view of the efficiency of the algorithms: the number of operations on abstract substitutions performed by the various algorithms. The results are summarized in Table 7.4 and depicted in detail in Tables A.1, A.2, A.3, A.4, A.5, A.6, A.7, A.8, A.9, A.10, A.11, A.12 given in the Appendix.

Table 7.4 contains, for each abstract operation on all benchmark programs, the number of calls in algorithms Original, Prefix and Caching. CA eval also gives the number of calls in Caching which are really evaluated (all the others being cached). Finally, it gives the percentage of operations saved for each of the improvements. Besides the traditional operations such as RESTRG and EXTG, results are also given for COMPARE (i.e. comparing two substitutions and returning *equal, smaller, greater,* or *not comparable*), SMALLER (i.e. testing if a substitution is smaller than another substitution), AI_TEST (i.e. the built-in arithmetic comparisons) and AI_IS (i.e. the function is of Prolog). Note also that operation EXTG is only performed for procedure calls and is integrated into operations UNIF_FUNC and UNIF_VAR for built-ins.

The ratio OR-PR indicates the percentage of calls saved for each of the operations by Prefix over the original algorithm. Half of the operations have a ratio of over 30% reaching peaks of about 39% and 41% for AI_TEST and AI_IS. The time consuming operations UNIF_VAR, UNIF_FUNC and EXTG dealing with unification achieve improvements of about 31, 34, and 22%.

The ratio OR-CA eval indicates the percentage of executed calls saved by Caching. These ratios are much higher than in the case of Prefix including peaks of 76 and 71% for COMPARE and EXTEND and 36, 41, and 33% on the unification operations. This seems to indicate and to confirm the results of our previous section that, the more costly the abstract operations, the more attractive will be Caching. When only unification instructions which are concerned (i.e. UNIF_VAR, UNIF_FUNC, EXTG) are considered, Caching produces a 7% improvement over Prefix and a 36% improvement over Original. Given the overhead for handling the caches, this fits nicely with the results observed for computation times.

The ratio OR-CA gives the number of calls to the operations spared by Caching. Caching basically calls the same operations as Original (but many of them are trivially performed through caching) except in the case where some operations are called inside operations. This is true for SMALLER and COMPARE where the number of calls is substantially reduced.

The lowest improvement occurs for EXTG which was to be expected since this is the instruction executed just after a goal. Each time the output of an abstract tuple has been updated, EXTG has to be evaluated. On the other hand, EXTEND has the highest improvement which is not surprising since this is the operation performed first when an abstract tuple is considered. The most important differences between Caching and Prefix appear in operations UNION and RESTRC, no difference occurring in EXTC. The last result is easily explained since different clauses very often have a different number of variables in their normalized versions. The former result is explained by the fact that Prefix has in fact little to offer for the above operations. For instance, RESTRC is only avoided when the whole clause is not reconsidered.

As far as the individual tables are mentioned, a few facts deserve to be mentioned. Read seems to be very peculiar, mainly due to the fact that the program is highly mutually recursive and that the domain is not particularly adequate for the program (see [15] for a discussion of this). As a consequence, it requires many iterations, exhibits excellent ratios for EXTEND, RESTRC, but rather lower improvements in general. Disj, on the other hand, has excellent ratios almost everywhere due to its tail-recursive nature (and its substitution-preserving property (see [14, 15])).

7.4 Time Distribution

In this section, we investigate the distribution of the computation time in various categories, including the abstract time (the time spent in the abstract operation), the control time (the total time - the abstract time), and the cache time (the time taken in managing the caches).

Table 7.5 describes the time distribution for caching. TT is the total time, TA the abstract time, TC the control time, and TH the cache time. TA is in fact a lower bound on the abstract time since an abstract operation is never reexecuted. Moreover, some of the operations (i.e. ADJUST and EXTEND) are not included. The reason is that, on

Program	TT	TA	TC	TH	TA%TT	TC%TT	TH%TT
Kalah	5.59	5.03	0.56	0.23	90	10	4
Queens	0.16	0.14	0.02	0.00	87	13	0
Press1	25.62	22.48	3.14	2.37	88	12	9
Press2	8.36	7.36	1.00	0.89	88	12	10
Peep	6.29	5.70	0.59	0.57	90	10	9
CS	5.92	5.60	0.32	0.32	94	6	5
Disj	3.25	3.00	0.25	0.25	92	8	7
PG	0.76	0.73	0.03	0.03	96	4	4
Read	25.37	23.73	1.64	1.40	93	7	5
Gabriel	2.06	1.80	0.26	0.16	87	13	8
Plan	0.60	0.52	0.08	0.02	87	13	3
Qsort	0.23	0.17	0.06	0.04	74	26	12

Table 7.5: Distribution of Computation Times for Caching

Program	TT	TA	TC	TA%TT	TC%TT
Kalah	9.22	8.89	0.33	96.42	3.58
Queens	0.24	0.22	0.020	91.67	8.33
Press1	38.21	37.44	0.77	97.98	2.02
Press2	11.58	11.47	0.11	99.05	0.95
Peep	7.17	7.15	0.02	99.72	0.28
CS	7.09	7.09	0	100.00	0.00
Disj	6.79	6.79	0	100.00	0.00
PG	1.07	1.07	0	100.00	0.00
Read	30.27	30.03	0.24	99.21	0.79
Gabriel	2.96	2.88	0.08	97.30	2.70
Plan	0.74	0.68	60	91.89	8.11
Qsort	0.34	0.32	20	94.12	5.88

Table 7.6: Distribution of Computation Time for `Original`

the one hand, these operations contain suboperations that are included, and, on the other hand, much of the remaining time is spent in the updating of the set of abstract tuples which is best considered as control. The ratios `TA%TT`, `TC%TT` and `TH%TT` give the percentage of the total time spent in the abstract time, the control time, and the cache time. The results indicate that about 90% of the time is spent in the abstract operations. `PG` and `CS` are the most demanding in terms of abstract time, which is easily explained as they manipulate large substitutions and make relatively few iterations (especially `CS`). The results also indicate that the cache time takes a significant part of the control time, including 10% on `Press2`. However, assuming a no-cost implementation of the control part, only about 10 % can be saved on the computation times. This indicates that the room left for improvement is rather limited.

Table 7.6 depicts the same results (except the cache time) for the original pro-

Program	TT	TA	TC	TA%TT	TC%TT
Kalah	6.46	6.21	0.250	96.13	3.87
Queens	0.15	0.12	0.030	80.00	20.00
Press1	29.4	28.82	0.430	98.03	1.97
Press2	8.85	8.45	0.400	95.49	4.51
Peep	5.86	5.74	0.12	97.95	2.05
Cs	5.87	5.67	0.2	96.60	3.40
Disj	3.03	3.03	0.00	0.00	0.00
Pg	0.86	810	0.05	94.19	5.81
Read	25.12	24.73	0.39	98.44	1.56
Gabriel	1.93	1.89	0.04	97.93	2.07
Plan	0.56	0,52	0.04	92.86	7.14
Qsort	0.23	0.2	0.03	86.96	13.04

Table 7.7: Distribution of Computation Time for Prefix

Program	SMALLER	AI-TEST	AI-IS	AI-VAR	AI-FUNC	EXTG	RESTRG	EXTC	RESTRC	UNION
Kalah	0.38	2.17	3.31	0.57	38.72	45.23	0.85	1.79	1.98	5.00
Queens	0.00	9.76	4.07	0.00	45.53	30.08	0.81	2.44	2.44	4.88
Press1	1.33	1.11	1.60	0.53	42.72	41.44	1.33	2.26	1.86	5.81
Press2	0.68	0.68	1.50	0.41	46.92	38.99	1.50	2.19	2.05	5.06
peep	0.18	0.26	0.09	7.50	53.09	25.31	1.23	2.65	3.70	6.00
CS	0.17	0.95	3.02	1.04	39.60	49.27	0.78	0.86	1.81	2.50
Disj	0.32	0.10	1.16	0.10	58.05	36.06	0.84	1.16	1.16	1.06
PG	1.37	0.14	4.93	1.37	41.37	40.14	1.51	2.47	2.33	4.38
Read	0.66	3.01	0.16	2.10	45.76	38.01	1.24	1.85	1.11	3.05
Gabriel	0.55	0.00	6.02	1.81	35.05	47.75	1.15	2.14	1.86	3.67
Plan	1.93	2.12	0.00	0.19	28.52	53.56	2.50	2.50	3.08	5.59
Qsort	0.00	1.23	0.00	6.17	24.69	56.17	1.85	2.47	2.47	4.94

Table 7.8: Percentage of Time Distribution Among the Abstract Operations in Caching

gram. It indicates that the control time is very low, only reaching 9 and 8 % for Queens and Plan but being lower than 3% in most cases. The negligible times for CS, Disj and PG may be explained by the fact that these programs are demanding in abstract time. Comparing those results with Caching, we observe that the control time in Caching has grown significantly due to the cache time (the rest of the control time being theoretically the same between Caching and Original).

Table 7.7 depicts the same results (except the cache time) for Prefix. It indicates, as expected, that the control times are almost always smaller then those of Caching and greater than those of Original. Also the control times are much closer to Original than to Caching,

Table 7.8 depicts the distribution of the abstract time among the abstract operations for Caching. It clearly indicates that the most time-consuming operations are UNIF_FUNC and EXTG confirming some of the results of the previous section. For

Program	SMALLER	AI_TEST	AI_IS	AI_VAR	AI_FUNC	EXTG	RESTRG	EXTC	RESTRC	UNION
Kalah	0.79	1.01	3.37	0.56	39.06	46.91	1.23	1.01	1.80	4.26
Queens	4.35	8.70	4.35	0.00	52.17	17.39	4.35	4.35	0.00	4.35
Press1	1.71	0.61	1.92	0.67	48.29	36.79	1.25	1.60	1.84	4.73
Press2	1.08	0.36	2.24	0.72	52.96	33.75	1.26	1.89	1.62	4.13
Peep	0.56	0.14	0.00	7.05	58.11	23.55	0.85	1.83	2.82	5.08
CS	0.26	0.51	3.32	0.77	48.59	41.05	0.64	1.28	1.41	2.17
Disj	0.29	0.00	1.75	0.15	67.45	27.45	1.02	0.73	0.44	0.73
PG	0.94	0.00	6.60	1.89	49.06	33.96	0.94	1.89	0.94	3.77
Read	0.87	2.11	0.20	1.91	49.41	37.76	1.51	1.98	1.21	3.05
Gabriel	0.70	0.00	7.37	3.16	41.75	39.30	1.05	1.75	2.11	2.81
Plan	1.52	1.52	0.00	1.52	36.36	46.97	3.03	1.52	3.03	4.55
Qsort	3.23	0.00	0.00	12.90	38.71	35.48	3.23	0.00	3.23	0.23

Table 7.9: Percentage of Time Distribution Among the Abstract Operations in Original

Caching, operations UNIF_FUNC and EXTG take more than 80% of the time except for Queens (75%). Operation UNION seems to be the next most demanding operation, but far behind the above two operations.

Table 7.9 depicts the distribution of the abstract time among the abstract operations for Original. The results indicate once again that the most time-consuming operations are UNIF_FUNC and EXTG. The results are also almost similar to those of Caching. Other operations have somewhat different ratios due to the fact that the unification takes most of the time.

7.5 Memory Consumption

Tables 7.10 and 7.11 depict the memory consumption of the three programs when bits and characters are used for representing the sharing component respectively. The before field gives the memory requirement before abstract interpretation, i.e. it includes the data structures necessary for parsing and compiling the Prolog programs as well as the sizes of the hash tables in the case of Caching. The max field gives the maximum memory requirement during the execution of the program. The most memory demanding program is Press1. It requires 279 kilobytes for Original, 1057 for Prefix and 2952 for Caching. In average, Prefix requires 2.35 more memory than Original but reaches peaks of 4.35 and 3.79 on Read and Press which are the most time consuming programs as well. Caching requires around 9.33 more memory than original in average and reaches a peak of 13.49 on Read.[1] Caching requires around 4 times as much memory as Prefix but the ratios are lower on the most demanding programs (2.79 on Press1 and 3.10 on Read.

When characters are used, Press1 requires 324, 1314, and 3831 kilobytes for

[1]The high ratios on Qsort and Queens are not significant since the initialization takes most memory.

Program	Original		Prefix	Caching		Pr/Or	Ca/Or	Ca/Pr
	before	max	max	before	max			
Append	2	4	5	148	152			
Kalah	56	125	244	204	723	1.95	5.78	2.96
Queens	6	12	18	152	182	1.5	15.1	10.11
Press1	82	279	1057	231	2952	3.79	10.58	2.79
Press2	84	176	450	233	1194	2.55	6.78	2.65
Peep	108	145	388	258	1197	2.67	8.25	3.08
CS	42	96	172	190	586	1.79	6.1	3.4
Disj	34	61	120	181	431	1.96	7.06	3.59
PG	13	32	61	159	272	1.9	8.5	4.45
Read	91	210	913	240	2834	4.35	13.49	3.1
Gabriel	26	57	123	173	412	2.16	7.22	3.34
Plan	16	35	66	163	247	1.88	7.05	3.74
Qsort	5	12	21	151	190	1.75	15.83	9.04
Mean						2.35	9.33	4.04

Table 7.10: Memory Consumption: Results with the Bit Representation of Sharing

Original, Prefix and Caching. In average, Prefix requires 2.46 more memory than Original but 4.85 and 4.05 more on Read and Press1. Caching requires 8.84 times as much memory as Original in average and reaches peaks of 16.11 on Read.[2] Caching requires 3.85 more memory than Prefix and 2.91 and 3.33 on Press1 and Read.

Table 7.12 depicts the percentage of memory saved by using bits instead of characters to represent the sharing component. The average saving are respectively 21, 22 and 19 % for Original, Prefix, and Caching.

7.6 Results on a Simpler Domain

In this section, we report some experimental results on a simpler domain, i.e. the mode domain of [20] which is a reformulation of the domain of [4]. The domain could be viewed as a simplification of the domain discussed so far where the pattern information has been omitted and the sharing has been simplified to an equivalence relation although all operations are in fact significantly different. The operations are much simpler but the loss of accuracy is significant. Nevertheless the efficiency results illustrate the potential of the improvements even in unfavorable conditions.

Tables 7.13 and 7.14 depict the efficiency results for the three programs with the bit and character representations of the sharing. For the bit version, Prefix reduces the computation by 28% compared to Original while Caching produces a 26% improvement. The improvements still remain significant, given that the improvements of Prefix and Caching on the sophisticated domain were respectively 28% and 31%. For the character version, there is now a much larger difference in efficiency between

[2]Note that the average is only better because of the initialization effect.

Program	Original		Prefix	Caching		Or/Pr	Or/Ca	Pr/Ca
	before	max	max	before	max			
Append	2	13	14	148	162			
Kalah	56	149	309	204	983	2.07	6.54	3.18
Queens	6	22	28	152	195	1.27	8.86	6.96
Press1	82	324	1314	231	3831	4.05	11.82	2.91
Press2	84	201	547	233	1525	2.72	7.58	2.78
Peep	108	157	453	258	1457	2.88	9.88	3.21
CS	42	121	234	190	860	1.93	7.10	3.67
Disj	34	70	156	181	578	2.22	8.25	3.70
PG	13	45	80	159	315	1.77	7	3.93
Read	91	237	1144	240	3820	4.82	16.11	3.33
Gabriel	26	70	150	173	502	2.14	7.17	3.34
Plan	16	46	82	163	275	1.78	3.97	3.35
Qsort	5	21	32	151	207	1.52	9.85	6.46
Mean						2.46	8.84	3.85

Table 7.11: Memory Consumption: Results with the Character Representation of Sharing

Program	Original	Prefix	Caching
append	69.23	64.29	6.17
kalah	16.11	21.04	26.45
queens	45.45	35.71	6.67
press1	13.89	19.56	22.94
press2	12.44	17.73	21.70
peep	7.64	14.35	17.84
CS	20.66	26.50	31.86
disj	12.86	23.08	25.43
PG	28.89	23.75	13.65
read	11.39	20.19	25.81
gabriel	18.57	18.00	17.93
plan	23.91	19.51	10.18
qsort	42.86	34.38	8.21
Mean	21.22	22.82	19.06

Table 7.12: Memory Consumption: Saving obtained by the Bit Representation

Program	OR	PR	CA	PR-OR	CA-OR
Append	0.02	0.02	0.04		
Kalah	2.60	1.81	1.88	0.30	0.28
Queens	0.18	0.14	0.15	0.22	0.17
Press1	6.08	4.08	4.26	0.33	0.30
Press2	6.17	4.23	4.31	0.31	0.30
Peep	5.54	3.86	4.03	0.30	0.27
CS	9.92	7.76	7.29	0.22	0.27
Disj	3.68	2.09	2.20	0.43	0.40
PG	0.48	0.38	0.40	0.21	0.17
Read	5.92	4.25	4.45	0.28	0.25
Gabriel	1.26	0.88	0.94	0.30	0.25
Plan	0.37	0.30	0.34	0.19	0.08
Qsort	0.32	0.23	0.20	0.28	0.37
Mean				28.21	25.91

Table 7.13: Computation Times and Percentages on the Small Domain: Bit Version

`Prefix` and `caching`. `Prefix` now brings around 29% improvement while `Caching` only improves `Original` by 6%. Note also that the computation times are significantly reduced compared to the sophisticated domain, all times being less than 8 seconds.

These results indicate the potential of the improvements even on small and simple domains. It also gives us a first confirmation that the simpler the abstract domain, the more interesting `Prefix` becomes.

Program	OR	PR	CA	PR-OR	CA-OR
Append	0.01	0.02	0.02		
Kalah	1.91	1.41	1.94	0.26	-0.02
Queens	0.15	0.08	0.14	0.47	0.07
Press1	4.72	3.10	4.12	0.34	0.13
Press2	4.74	3.19	4.37	0.33	0.08
Peep	4.22	2.91	4.04	0.31	0.04
CS	7.40	5.83	6.95	0.21	0.06
Disj	2.72	1.57	2.27	0.42	0.17
PG	0.39	0.29	0.39	0.26	0.00
Read	4.52	3.28	4.42	0.27	0.02
Gabriel	0.96	0.69	0.93	0.28	0.03
Plan	0.29	0.25	0.29	0.14	0.00
Qsort	0.25	0.19	0.21	0.24	0.16
Mean				29.45	6.15

Table 7.14: Computation Times and Percentages on the Small Domain: Character Version

Chapter 8

Widening

In this chapter and the next one, we will try two other techniques. Contrary to both previous methods, these ones are not optimizations. They are two experimentations that must lead to improvement of the precision of the results. The first one is a mean to detect strictly increasing chains. The second one is reexecution of the goals of a clause. In this chapter, we present the ideas underlying the widening. But, this technique alone is not sufficient to obtain an interesting gain. So, it will be used together with reexecution, in the next chapter.

The widening is a method to insure the termination of the algorithm. Indeed, if the domain is finite, in the worst possible case, all the possibilities must be examined, but it can always be done, soon or later. But, if the domain is infinite, there can arise a process of creation of a term of strictly increasing length that would inevitably lead to an infinite loop. The principle used is the one presented by P. Cousot and R. Cousot in [7]. It consists in widening the doubtful substitutions in some less precise ones with no more strictly increasing chains.

For a chain to increase strictly, it must be each time more and more precise, that is, each time smaller and smaller. So, if the input substitution for a new call is made greater or equal than the input substitutions of all the previous calls, there can not be any infinite loop. But the greater the substitution, the more important the loss of precision. Hence, we need the smaller abstract substitution which is greater than both the new one and the greatest of all the previous ones. That is, we need the `UNION` of these two substitutions.

Definition 18 The widening is a function $W : AS_D \times AS_D \rightarrow AS_D$, D being a set of program variables, satisfying:

- $\forall$ sequence $\beta_1, \beta_2, \ldots, \beta_i, \ldots (\beta_k \in AS_D)$, the sequence $\alpha_1, \alpha_2, \ldots, \alpha_i, \ldots$, $\alpha_1 = \beta_1$, $\alpha_{i+1} = W(\alpha_i, \beta_{i+1})$, is not decreasing;
- $\forall \beta_1, \beta_2 \in AS_D$, $\beta_1 \leq W(\beta_1, \beta_2)$ and $\beta_2 \leq W(\beta_1, \beta_2)$.

We will examine two different widening methods. A first primitive one (presented in [15]) and then a more clever one. As the *mode* domain does not take care of

structures (just *Ground*, *Variable* or *Any*, no matter the form of the terms), it is a finite domain and this experimentation is not relevant for it. It just concerns the *type* domain. Firstly, for a better understanding, we shall present the *suspended* stack. Then, the original implementation of the widening and the new one. And finally, a comparison between both methods.

8.1 The *Suspended* stack

The *suspended* stack contains a number of pairs (β, p) for which execution is suspended. Namely, pairs for which consideration has begun and for which the consideration of other pairs is needed. Their evaluation is suspended the time necessary for the execution of these other pairs. For example, consider the following predicate:

$$p(\bullet) : - \ldots, q_1(\bullet), \ldots, q_2(\bullet), \ldots$$

where "..." represent built-ins. When it is examined with input substitution β, it needs the result of (β', q_1) and of (β'', q_2) where β' is the result of the first built-ins and β'' is the result of all the preceding goals. The execution of (β, p) is suspended a first time during evaluation of (β', q_1) and then a second time during evaluation of (β'', q_2). Thus, during these two evaluations, the pair (β, p) belongs to the *suspended* stack. But, outside of those calculations, (β, p) can not be in the stack. Let us note that, although useless, (β, p) can be in the stack during evaluation of built-ins too. It is implemented that way for simplicity. Indeed, we then have to add (β, p) to the stack only once at the begining of its computation and remove it once at the end.

When a predicate is called, the stack is examined in search for it, the last entered element the first examined. If it is found, that means the call is a recursive one that will definitely lead to an infinite loop. But, if the input substitution is widenned, it is either equal to a previous suspended one or greater than all the previous ones. If equal, the execution is avoided and the temporary result is gotten back. If greater, we know there is no problem and we can go on with the execution of the call.

8.2 The original widening

The original widening, presented in [15], is a very simple one. It consists in systematically widening all input substitutions of recursive calls.

Let us examine a recursive call of a procedure p. Let us denote β_1 the greatest substitution so far in input of p and β_2 the input substitution of the new call. A simple comparison can be performed between them. Different cases may arise:

- $\beta_2 = \beta_1$: the `UNION` is the same. As (β_1, p) is suspended, so is (β_2, p) which is not refined.
- $\beta_2 < \beta_1$: the `UNION` is β_1 and (β_1, p) is suspended, so not examined.

- $\beta_2 > \beta_1$: the UNION is β_2 and (β_2, p) is examined.
- otherwise: the UNION must be computed and $(\texttt{UNION}(\beta_1, \beta_2), p)$ is examined.

Note that β_1 the greatest of all (β_i, p) in the stack is easy to find. Indeed, as pairs appended to the stack are always computed as a UNION of the previous ones, the greatest is surely the last one with predicate p in the stack.

8.3 The new widening

The new method is a slightly more complicated one. Its purpose is to widen substitutions only when it is necessary. The basic idea is to detect the creation of stictly increasing structures and to widen only substitutions taht contain such structures.

Definition 19 A term t is an instance of a term t' iff

- neither t nor t' have a form, that is, we know nothing more than their type,
- t is of the form $f(\ldots)$ and t' has no form,
- or t and t' both have the same functor, the same arity and all subterms of t are instances of their corresponding subterm in t'.

For example, let $t = f(X)$ and $t' = f(g(X))$. Then, t' is an instance of t because they both have the same functor f of arity 1 and X is an instance of $g(X)$ because X has no form and $g(x)$ has a functor.

Definition 20 A term t is said to be a part of a term t' iff

- t' is an instance of t,
- or t' is of the form $f(t'')$ where f is any functor and t is a part of t''.

Definition 21 Let us denote $\beta_1, \ldots, \beta_k$ the input substitutions of all the previous calls to the procedure p and β the new one. Then, the creation of a strictly increasing structure occurs iff, $\exists l, 1 \leq l \leq k$ such that for each binding of β_l (say $X \mid t$) that is different of its counterpart in β (say $X' \mid t'$), t is a part of t'.

If a strictly increasing structure creation is detected with that test, then an infinite loop can be avoided by widening the substitution and examining $(\texttt{UNION}(\beta_l, \beta), p)$ instead of (β, p). But, if no strictly increasing structure is detected in the input substitution of a recursive call, then no widening is done.

Note that, with this test, all $(\beta_i, p), 1 \leq i \leq k$ may be examined if necessary and not just the pair with the greatest β_i. So, the *suspended* stack could just be seen as a *suspended* set.

8.4 Comparison between both tests

The big difference between those two methods is a gain in precision. Indeed, in the original one the `UNION` operation is computed more often than in the new one. It is due to the fact that it is done, in the second method, only when a strictly increasing structure creation is detected but every time in the original method.

For instance, let us examine[1] the execution of the test QSORT (figure 8.1) already used in the previous chapters. Let us assume it is called `qsort/2`$(Ground(1), Var(2))$. Then, `qsort/3` is directly called with

$$\beta_1 = \{(Ground(1), Var(2), Ground(3) : []\}$$

which is examined. Later on, a first recursive call to `qsort/3` occurs with

$$\beta_2 = \{Ground(1), Var(2), Var(3)\}$$

With the original method, widening occurs and

$$\texttt{UNION}(\beta_1, \beta_2) = \{Ground(1), Var(2), GV(3)\}$$

is used instead of β_2. But, with the new method, as β_1 is not part of β_2, no widening is necessary and `qsort/3` is examined with β_2. In fact, the final result is exactly the same with both methods. But, it may become very interresting if we examine all elements in the *sats* and use them to do specialisations.

[1] We ask the reader to believe what we say. The length of the verbose of execution of that test make it impossible to include it in this report.

```
qsort(S,Sorted) :-
   qsort(S,Sorted,[]).

partition([],F,[],[]).
partition([F|T],P,[F|S],B) :-
   F <= P,
   partition(T,P,S,B).
partition([F|T],P,S,[F|B]) :-
   F > P,
   partition(T,P,S,B).

qsort([],X,X).
qsort([F|T],Res,Tail) :-
   partition(T,F,S,B),
   qsort(S,Res,Others),
   Others = [F | Rest],
   qsort(B,Rest,Tail).

split(T,F,T,T).
```

Figure 8.1: QSORT

Chapter 9

Reexecution

The "*Reexecution*" aims to improve the precision of the results. The chosen technique is to transform the program[1] to another one whose concrete semantics is unchanged with respect to the SLD-resolution. However, this transformation reorganizes the program atoms so that the algorithm may find a more accurate result (a smaller fixpoint). In the next sections, we first explain the transformation, next the algorithm for the *mode* domain and for the *type* domain. The last section discusses some results (cpu-time,memory,...) about the two implementations.

This work was realized at last so this one was partially achieved. We present here the first results.

9.1 The transformation

As transformation, we choose to place a copy of some atoms at the end of the clauses. It doesn't change the semantics of the programs as proven in the following proposition:

Proposition 22 Let P be a normalized Prolog program. $P = \{C_i\}$ where C_is are the clauses of P.
If $C_i \equiv p(X_1, \ldots, X_n) : -g_1, g_2, \ldots, g_m$
If $P' = P\langle C_i \longrightarrow C_i' \rangle$
where $C_i' = g_1, g_2, \ldots, g_m, g_{i_1}, \ldots, g_{i_r} \quad \forall\, i_1, \ldots, i_r \in \{1, \ldots, m\}$
Then P and P' have the same semantics.
Proof We know that the semantics computed as the least fixpoint of the transformation TSCT is equivalent to the SLD-resolution [13, pp 9–10]. Since the SLD-resolution doesn't depend on the computation rule [19, pp 49–55], hence C_i' can again be rewritten as:

$$C_i'' \equiv p(X_1, \ldots, X_n) : -a_1, \ldots, a_m.$$

[1] In this section, we designate the Prolog program by the word "*program*" and the abstract interpreter by "*algorithm*".

where $a_j = g_j, g_j$ if $i \in \{i_1, \ldots, i_r\}$
$a_j = g_j$ otherwise.

For example:

$C_i \equiv p(\bullet) :- g_1, g_2, g_3.$
$C_i' \equiv p(\bullet) :- g_1, g_2, g_3, g_2.$
$C_i'' \equiv p(\bullet) :- g_1, g_2, g_2, g_3.$

$$C_i'' \equiv p(\bullet) :- g_1, \alpha, g_2, \beta, g_2, \gamma, g_3.$$

Let be β the result substitution after the call to g_2 with the input substitution α. It's obvious that the second call to g_2 with the same argument will be successful. Hence, β is equal to γ and we may omit the second occurrence of g_2. By generalization to the other cases, we can see that C_i'' is equivalent to C_i.□

Corollary 1 *Let $P = \{C_i\}$ be a program and its clauses.*
Then the semantics of P is unchanged if some atoms are repeated many times and anywhere in a clause.

In this implementation, the Prolog program isn't transformed directly as depicted previously. The algorithm is adapted to behave as if the program is modified. In the original algorithm, before calling the function `RestrC`, `SolveClause` has already treated each atom once. So it's sufficient to reexecute the interesting atoms. This operation is done by calling the function `ForwardPropagate` just before the call to `RestrC`.

We have not yet explained why this technique improve the results quality, or how this technique is implemented. These aspects are covered by the next sections.

9.2 Implementation for the *mode* domain

9.2.1 Algorithm

The basic idea for the *mode* domain is to reexecute each atom until the current substitution can't be improved any more. At a given time, during the reexecution, the substitution can lose precision. So, to avoid this, we take systematically the *glb* [2] of the substitution computed until now and the one obtained after the last goal

[2] *glb* is the function "greater least bound" defined as:

$glb(\alpha, \beta) = \gamma \iff \gamma \leq \alpha$ and $\gamma \leq \beta$
and $\theta \leq \alpha \Longrightarrow \theta \leq \gamma$
and $\theta \leq \beta \Longrightarrow \theta \leq \gamma$

```
procedure SolveClause( in β_in,p,c,suspended; out β_out; inout sat,dp)
begin
    β_ext := EXTC(c,β_in);
    for i := 1 to m with b_1,...,b_m body-of c do
    begin
        β_aux := RESTRG(b_i,β_ext);
        switch (b_i) of
        case X_j = X_k:
            β_int := AI_VAR(β_aux)
        case X_j = f(...):
            β_int := AI_FUNC(β_aux,f)
        case q(...):
            SolveGoal(β_aux,q, suspended,sat,dp);
            β_int := sat(β_aux,q);
            if (β_in,p) ∈ dom(dp) do
              ADD_DP(β_in,p,β_aux,q,dp)
        end;
        β_ext := EXTG(b_i,β_ext,β_int)
    end;
    ForwardPropagate(β_in,β_ext, sat, dp, c, p, suspended)
    β_out := RESTRC(c,β_ext)
end
```

where $\mathcal{V}(g)$ is the set of all the variables encountered in the goal g. For example: $\mathcal{V}(r(X_1, X_5, X_3)) = \{X_1, X_3, X_5\}$.

Figure 9.1: A. *mode* ForwardPropagate's Algorithm

encountered. We have, in fact, to chose between two abstract substitutions. The best solution is to take the *glb* of both. So, we extract the best information of both substitutions in a new one. The algorithm of `SolveGoal` is unchanged. The reader can find the algorithm of `ForwardPropagate` in the figures 9.1, 9.2, 9.3.

The dependence of the goals on the variables permits to reconsider some goals when their arguments have been modified. When the domain is finite, it's easy to prove that the dependence isn't a vicious circle.

Proposition 23 The function `ForwardPropagate` ends if the domain is finite.

Proof

Let i be the index of β_{ext} at the point $\Diamond 1$.

The demonstration is made by induction on the value of β_{ext} at the point $\Diamond 1$. The value of β_{ext} decreases each time the point $\Diamond 1$ is reached. A finite domain insures us then that the program ends. Otherwise:

$$\exists j \ : \ \beta_{\text{ext}_j} \geq \beta_{\text{ext}_{j+1}}$$

```
procedure ForwardPropagate(in: β_in1, inout: β_in2, sat, dp
in: C, p, suspended)
%% let C ≡ p(X_1, ..., X_n) :- g_1, ..., g_m.

begin
    S := ∪_{i=1,...,m} V(g_i)
    β_ext := β_in2
    while β_ext ≠ ⊥ and S ≠ ∅ do
    begin
        {◇1}  E := ∅
              for i = 1 to m do
              begin
                  if V(g_i) ∩ S ≠ ∅ do
                  begin
                      Call(β_in1, β_ext, g, suspended, p, dp, sat, β_out)
                      β_f := glb(β, β_out)
                      E := E ∪ {X : X is a variable ∧ Xβ_f ≠ Xβ_ext}
                      β_ext := β_f
                  end
              end
        {◇2}  S := E
    end
    β_in2 := β_ext
end
```

Remark: $X\alpha \neq X\beta$ denotes that the mode of variable X is different between the both abstract substitutions α and β.

Figure 9.2: B. *mode* ForwardPropagate's Algorithm

procedure Call(**in:** $\beta_{\text{in}}, \beta_{\text{ext}}, g, suspended, p$ **inout:** dp, sat, **out:** β_{out})

begin
  β_{aux} := Restrg(g, β_{ext})
  switch (g) **of**
    case $X_j = X_k$: β_{out} := AI_VAR(β_{aux})
    case $X_j = f(\ldots)$: β_{out} := AI_FUNC(β_{aux}, f)
    case $q(\ldots)$: SolveGoal($\beta_{\text{aux}}, q, suspended, sat, dp$)
      $\beta_{\text{out}} := sat(\beta_{\text{aux}}, q)$
      if $(\beta_{\text{in}}, p) \in dom(dp)$ **do**
        ADD_DP($\beta_{\text{in}}, p, \beta_{\text{aux}}, q, dp$)
  β_{out} := EXTG($g, \beta_{\text{ext}}, \beta_{\text{out}}$)
end

Figure 9.3: C. *mode* ForwardPropagate's Algorithm

Since the *glb* operation insures us that $\beta_{\text{ext}_{j+1}}$ is less or equal to β_{ext_j}, thus $\beta_{\text{ext}_{j+1}} = \beta_{\text{ext}_j}$. Between the j^{th} and $(j+1)^{\text{th}}$ loop, the algorithm is passed througout the point $\diamondsuit 2$. Hence $S_{j+1} = \emptyset$ and the program ends. □

9.2.2 Application

To understand why the *reexecution* technique improves the results with the *mode* domain, let us see a small example.

$p(X_1) : -X_1 = [X_2 \mid X_3], X_2 = 1, X_3 = 2.$
$?p(var) \longrightarrow p(ground)$

For this program, the original *mode*-algorithm can't remember that X_2 and X_3 are both components of X_1. For this reason, this one can't deduce that X_1 is *ground* and returns *any*. With the reexecution, the *mode*-algorithm can be as good as the *type*-algorithm for this particular case. This technique corrects the *mode* domain "defect" brought to the fore by K. Musumbu in [20, pp II/53] .

9.3 Implementation for the *type* domain

9.3.1 Algorithm

The technique for the *type* domain is the same as for the *mode* domain except a few points:

- The *type* domain is sufficiently precise to neglect built-ins during reexecution. In fact only calls executed with arguments become more precise after the computation must be reexecuted.
- The *type* domain isn't finite. The algorithm can fall in a vicious circle by attempting to gain more and more precision on a variable as shown in figure 9.4. For this reason, a mechanism must avoid such situations.

X/f variable → *X/f f variable* → *X/f f f variable* → *f f f f ...*

Figure 9.4: A vicious circle

The *type* reexecution algorithm is described in figures 9.5, 9.6. To prevent a vicious circle due to infinite dependence[3], a stack memorizes all the input substitutions of calls. Before computing a call, the algorithm checks if widening occurs between the input substitution and one from the stack (function `Detect`). If an infinite structure is detected, the call is skipped and forgotten until the end of reexecution.

9.3.2 Application

In the following program,

$p(X_1, X_2) : -g(X_1, X_2), X_1 = [].$
$g(X_1, X_2) : -X_1 = 1, X_2 = [X_3 \mid X_4].$
$g(X_1, X_2) : -X_2 = 1.$

when g is called with $\{X_1/var(1), X_2/var(2)\}$, the abstract interpreter doesn't know again that only one of both clauses is interesting (the second one because the first one should fail). Thus, the result of $g(var(1), var(2))$ is $g(Gv(1), Novar(2))$ which is the `UNION` of the results of both clauses. But when the `UNION` is computed, precision is lost! The reexecution allows to remember that X_1 is *ground* and has a functor [] different from 1 when g is called for a second time. So only the second clause is examined and precision is gotten back. The result is $\{X_1/Ground(1) : [], X_2/Ground(2) : 1\}$.

9.3.3 Another Strategy

We have introduced the reexecution without special attention to the order used to choose the goals. In this strategy, all the goals are reexecuted by entire sequence. Another possibility is to reexamine each goal since the first clause's goal each time a goal is reexecuted. The following example illustrates both techniques:

[3] a variable becoming more and more precise

```
procedure SolveClause( in β_in,p,c,suspended; out β_out; inout sat,dp)

begin
    ... exactly the same algorithm as for the mode domain ...
end


ForwardPropagate(in: β_in1, inout: β_in2,sat,dp, in: C,p,suspended)


%% let C ≡ p(X_1,...,X_n) :- ...,g_1,...,g_m,....
%% g_is are all the calls of this clause.
%% The "..." represent built-ins.
begin
    S := ∪_{i=1,...,m} V(g_i)
    stack := ∅
    R_i := true  ∀ i ∈ {1,...,m}
    β_ext := β_in2
    while β_ext ≠ ⊥ and S ≠ ∅ do
    begin
        E := ∅
        for i = 1 to m do
        begin
            if V(g_i) ∩ S ≠ ∅ and R_i do
            begin
                if Detect(g_i,β_ext,stack) do
                    R_i := false
                else
                    push(g_i,β_ext,stack)
                    Call(β_in1,β_ext,g,suspended,p,dp,sat,β_out)
                    β_f := glb(β,β_out)
                    E := E ∪ {X : X is a variable ∧ Xβ_f ≠ Xβ_ext}
                    β_ext := β_f
            end
        end
        S := E
    end
    β_in2 := β_ext
end
```

Figure 9.5: A. *type* ForwardPropagate's Algorithm

procedure `Call`(**in:** $\beta_{\text{in}}, \beta_{\text{ext}}, g, suspended, p$ **inout:** dp, sat, **out:** β_{out})

begin
β_{aux} := `Restrg`(g, β_{ext})
$q(\ldots)$: `SolveGoal`$(\beta_{\text{aux}}, q, suspended, sat, dp)$
$\beta_{\text{out}} := sat(\beta_{\text{aux}}, q)$x
if $(\beta_{\text{in}}, p) \in dom(dp)$ **do**
`ADD_DP`$(\beta_{\text{in}}, p, \beta_{\text{aux}}, q, dp)$
β_{out} := `EXTG`$(g, \beta_{\text{ext}}, \beta_{\text{out}})$
end

function `Detect`(**in:** $g, \beta, stack$)$\longrightarrow$ Boolean

pre: $\emptyset$
post: return *true* $\Longleftrightarrow \exists\ (g, \gamma) \in stack : \beta$ is an instance of *gamma*

Figure 9.6: B. *type* ForwardPropagate's Algorithm

old version: $\underbrace{g_1, g_2, g_3, g_4}_{\text{SolveClause}}, \underbrace{g_1, g_2, g_3, g_4}_{\text{first reexecution}}, \underbrace{g_2, g_3}_{\text{second}}, \underbrace{g_2}_{\text{third}}$

new version: $\underbrace{g_1, g_2, g_3, g_4}_{\text{SolveClause}}, \underbrace{g_1, g_1, g_1}_{g_1}, \underbrace{g_2, g_1, g_2}_{g_2}, \underbrace{g_3}_{g_3}, \underbrace{g_4, g_1, g_4}_{g_4}$.

Each time reexecution computes a goal, all the previous ones are again examined if they share a modified variable with this one until the current substitution becomes stable. Next, reexecution can examine the next goal and so on.

As we have previously explained, our work was unachieved. This new strategy was not implemented so we will not discuss its results here. However, at the print time, Pascal Van Hentenryck has completely implemented that version and he discusses the results in [16]. We can already say that the new version is much faster that the old one with the biggest programs (2 to 11 times faster) and gains a noticeable accuracy with the programs `PG` and `Press1`. However, with smaller programs, the old version seems to be faster but the computation times is only of a few seconds.

9.4 Experimental Evaluations

This section shows the results of the *reexecution*-algorithm. This one is the *caching*-version updated with the modifications described above. The `Read`

program can not be tested with the *type* domain because of a lack of memory [4]

9.4.1 Time Distribution

Figures 9.2 and 9.3 indicate computation times for the *mode* domain and the *type* domain. In the first table, programs `Press1`, `Press2` and `Read` are the slowest with 23, 22 and 18 sec. In the second one, there are `Kalah`, `press2` and `CS` with 100, 80 and 42 sec.

The table 9.1 shows the ratio between computation times for original and *reexecution-* algorithms. Ratios are majored by a factor 5 in the *mode* domain whereas in the *type* domain, ratios reach a peak of 18. It is probably due to the fact that *type* domain can gain in precision by two ways: either by changing variables mode or by modifying the pattern associated to a variable. In the first case, let us remember that the *type* domain has many more possible modes than the *mode* domain.

	Kalah	Queens	Press1	Press2	Peep	CS
Mode	3.03	1.13	5.38	5.39	2.06	4.87
Type	17.91	2.94	1.63	5.07	2.90	13.60

	Disj	Pg	Read	Gabriel	Plan	Qsort
Mode	3.28	5.02	4.26	3.22	1.50	4.05
Type	3.64	14.26	—	3.58	2.80	4.30

Table 9.1: Ratios between reexecution algorithm computation times for *mode* and *type* domains

[4] With the second strategy discssed above, the program `Read` is exected in less than 5 mintes.

Program	TT	TA	TC+TH	TT%TA	TT%TC+TH
Kalah	5.710	4.950	0.760	86.69	13.31
Queens	0.170	0.132	0.038	77.65	22.35
Press1	22.940	18.950	3.990	82.61	17.39
Press2	23.260	19.210	4.050	82.59	17.41
Peep	8.320	7.210	1.110	86.66	13.34
CS	35.520	33.720	1.800	94.93	5.07
Disj	7.210	6.720	0.490	93.20	6.80
PG	2.010	1.665	0.345	82.84	17.16
Read	18.950	15.390	3.560	81.21	18.79
Gabriel	3.030	2.409	0.621	79.50	20.50
Plan	0.510	0.388	0.122	76.08	23.92
Qsort	0.810	0.623	0.187	76.91	23.09

TT comptation time
TA comptation time or only abstract operations
TC+TH coptation time or the control algorithm and caching mechanism
unit: sec.

Table 9.2: Computation time for abstract interpreter with reexecution on the mode domain and *chars*

Program	TT	TA	TC+TH	TT%TA	TT%TC+TH
Kalah	100.100	95.320	4.780	95.22	4.78
Queens	0.470	0.330	0.140	75.00	25.00
Press1	41.900	36.480	5.420	87.06	12.94
Press2	42.420	37.370	5.050	88.10	11.90
Peep	18.260	16.550	1.710	90.64	9.36
CS	80.510	76.110	4.400	94.53	5.47
Disj	11.830	10.820	1.010	91.46	8.54
PG	10.840	9.410	1.430	86.81	13.19
Read	*not enough memory*				
Gabriel	7.370	6.440	0.930	87.38	12.62
Plan	1.660	1.444	0.216	86.99	13.01
Qsort	0.990	0.789	0.201	79.70	20.30

TT comptation time
TA comptation time or only abstract operations
TC+TH coptation time or the control algorithm and caching mechanism
unit: sec.

Table 9.3: Computation time for abstract interpreter with reexecution on the type domain and *chars*

9.4.2 Memory Consumption

Tables 9.4 and 9.5 depict the memory consumption with the "reexecution" version. This algorithm sometimes requires up to 10 times more memory than the orignal algorithm with caching. The *bit*-implementation is very interesting for the bigger programs, so 42, 38 and 34% are gained with programs `CS`, `Kalah`, and `Disj`. The more demanding programs are `CS`, `Press2`, `Press1` and `Kalah` with 7316, 7050, 6942 and 5518 Kb (for *char*-implementation) and 4329, 5168, 5968 and 3444 Kb (for *bit*-implementation).

The memory amount after compilation (*before* column in tables) is higher for the *mode* domain. Indeed, this one requires to stock the variables dependance for all the goals whereas *type* domain necessitates dependance only for calls.

It is now necessary to precise some items:

1. We mean by "memory consumption" the space necessary to execute the algorithm with *caching*-mechanism. So an important memory amount is used to optimize the *memoization* performances. This space is not strictly necessary to the algorithm execution. So amounts in the tables are overestimations.

2. The memory amounts are computed as the sum of all the algorithm requests (`malloc()`). So the memory required to manage the dynamic allocation is not posted.

9.4.3 Results Quality

The aim of reexecution is to improve the results quality. Let us see if this one is really improved by this technique. The global quality is computed as the *UNION* of all the input substitutions for a same predicate found in foundations. Indeed, this information can be directly used in a optimized interpreter described in page 21. So it is pertinent to compare these results between different versions.

comparison between *original*-version and *reexecution*-version

Table 9.6 depicts improvement of quality for *mode* and *type* domains due to the *reexecution* technique. The only possible enhancement for the *mode* domain is a *any* becoming *ground*. As early described, the *type* domain can improve its substitutions in two ways: the mode and the pattern associated to each variable. The improvement essentially concerns the first possibility: the mode component.

The *mode* domain takes the best advantage of the *reexecution* technique. A lot of results are unchanged in the *type* domain. How is it possible? In fact, we have only examined the input substitutions of calls, many result

substitutions may have been improved but being pure result (a substitution which is returned without being used as an input later), these ones are not discussed here. Other results do not belong to foundations, so they are not examined.

Comparison between *mode-reexecution* and *type-original*

Our aim is to see if the *mode-reexecution* algorithm (`MRA`[5]) is as good as *type-original* algorithm (`TOA`[6]) with respect to results quality and computation times. This comparison is suggested by K. Musumbu in [20, pp II/53]. The table 9.7 shows that `MRA` can sometimes be better in quality than `TOA` but with questionable performances.

[5] *mode* algorithm with reexecution implemented with chars and caching
[6] *type* algorithm without reexecution implemented with chars and without caching

	modes		types	
Program	**Before**	**After**	**Before**	**After**
Append	333	257	330	376
Kalah	589	918	478	5518
Queens	354	386	341	466
Press1	798	1918	578	6942
Press2	814	1941	586	7050
Peep	993	3103	677	4037
CS	670	2079	519	7316
Disj	516	964	429	1936
Pg	383	531	359	2022
Read	819	2672	void	void
Gabriel	463	770	400	1532
Plan	414	457	373	704
Qsort	348	416	338	520

Before: memory consumption just after compilation
After: memory consumption after execution
unit: KiloByte

Table 9.4: Memory consumption for the reexecution version with char implementation

	modes		types	
Program	**Before**	**After**	**Before**	**After**
Append	333	354	330	341
Kalah	589	716	478	3444
Queens	354	380	341	420
Press1	798	1723	578	5968
Press2	814	1741	586	5168
Peep	993	2395	677	3277
CS	670	1240	519	4239
Disj	516	663	429	1289
Pg	383	458	358	1470
Read	819	2215	592	void
Gabriel	463	675	400	1177
Plan	414	449	373	603
Qsort	348	389	338	460

Before: memory consumption just after compilation
After: memory consumption after execution
unit: KiloByte

Table 9.5: Memory consumption for the reexecution version with bit implementation

	Kalah	Queens	Press1	Press2	Peep	CS
Mode	♣	♣	♠	◇	♣	♣
Type	◇	◇	♣	◇	♠	♡

	Disj	Pg	Read	Gabriel	Plan	Qsort
Mode	♣	♠	♠	◇	♣	♣
Type	♡	♡	—	♠	◇	♠

♣ : a lot of *any*, *ngv*, ... are become *ground*.
♠ : quality doesn't change a lot.
◇ : quality is unchanged.
♡ : all was already *ground* in the original version

Table 9.6: Quality with/without *reexecution* for *mode* and *type* domains

Program	Quality	Comment	Ratio	Valuation
Kalah	$\simeq$	*any* s are GV in type	1.61	better
Queens	$=$	all is *ground*	1.29	better
Press1	$\simeq$	*any*s is NOVAR in type	1.65	better
Press2	$<$	a lot of GROUND are *any* in mode	0.50	very bad
Peep	$>$	2 *ground* are NOVAR in type	0.86	
CS	$<$	2 *any* are GROUND in type	0.22	very bad
DS	$=$	all is *ground*	0.94	bad
PG	$<$	9 *any* are GROUND in type	0.55	very bad
Gabriel	$\simeq$	*any* s become NOVAR in type	0.95	bad
Plan	$\simeq$	1 *any* is NGV in type	1.39	better
Qsort	$>$	1 *ground* is ANY in type	0.42	

Quality : comparison between *mode* domain's quality and *type* domain.
Ratio : $\frac{\text{TOA's time}}{\text{MRA's time}}$

Table 9.7: Comparison between the quality of MRA and TOA

Chapter 10

What could be done in the future?

From the starting Pascal program, we did so far a series of optimizations that ended with different C programs, all of them faster and less memory consuming or more precise. All these results are interesting, but most of them can be improved again. Here are a few ideas that could be implemented in the future.

10.1 Memory management

First of all, an improvement could be done in memory management. The standard ANSI "alloc" functions are good in the way memory is allocated, but they are slow. The better memory management functions we used are faster, but they allocate too much memory and a big amount of it is kept unused. Indeed, a memory manager has a large amount of memory at its disposal. When a request for memory arrives from the program, the memory manager must allocate a part of its ressource that has the requested size. And it must also get that memory back when the program has no use of it any more. So, the memory manager must have efficient procedures to handle all the requests and there are choices to do. If we want the memory management to take care of the least byte, it must be complicated and so slow. But, if we want it to be fast, a simpler management may be done (for instance, the memory manager may allocate an amount of memory bigger than the requested one that is a multiple of a ceratin basic amount) with memory left unused as a result.

The problem with those two methods is that they are two general methods that may suit to any program. With a good study of the problem, it may be possible to create memory allocation procedures more suitable to our needs. For instance, the dependency graph is made of a lot of elements which all have the same structure, and so the same size; they are allocated and disposed of

a lot of times. It could be a good idea to handle a stack of free cells. But the problem is much more important with the substitutions that are much bigger, and of changing size.

The garbage collector can be improved too. This one allows us to gain a big amount of memory but this one is fragmented in the sense that it is composed of a lot of small parts. But the GC can not use the fact that two contiguous free parts is in fact another bigger free one. Sometimes (especially in the *type* domain) abstract substitutions gain more and more precision and thus more and more memory. Although all old substitutions are destroyed, a memory problem can occur. A way to solve it, is to copy all used memory parts in another place in a contiguous way (secondary memory can eventually be used if there is not enough memory to do that).

A lot of times, `malloc()` is used to create local object inside functions. These ones are destroyed systematically when the execution of the function is over. In fact a stack [1] is a natural way to manage such objects. It could be a good idea to proceed such a way both to gain time, memory and to limit memory fragmentation.

10.2 Concurrent programming

In procedure `SOLVE_GOAL`, when a predicate is examined, all its clauses are examined, each at its turn, and the `UNION` of their results is computed. It might be a good idea to examine all the clauses in parallel on different processors. But, this could raise new problems. For instance, suppose a predicate is called and it is suspended. Does it mean that predicate is a recursive one for which a new subcomputation must not be started? Or does it mean the predicate (recursive or not) is already under examination from another clause; in that case the new subcomputation must be undertaken if inputs are different? Another problem is the sharing of global objets like lattices, Hasse diagrammes, dependency graphs.

These are a few ideas, but there may be a lot of others...

[1] Although each object has to be created by calling a function, all the local objects of a function are destroyed at once

Chapter 11

Conclusion

Abstract Interpretation, as we had already said, is an important tool to analyse statically Prolog programs and to improve performances of compilers. A basic interpreter had been developped by Pascal Van Hentenryck. This one was written in Pascal, it took a lot of time (a average time of 22sec on our panel of tests) and consumed a very big amount of memory (several megabytes with a peak of 20 megabytes for some programs). This program used the type domain defined in [20].

Our work was primarily to rewrite this interpreter in C. This language has a lot of interesting particularities which allow to improve general performances. This phase allows us to reexamine some crucial points of the algorithm like lattices, abstract unification, transitive closure, ... So, we have obtained a basic version of the algorithm in C wich was already faster than the old one. This version was yet improved in two ways:

- **caching** applies the memoization principle on all important abstract operators. So, a lot of computations were avoided.
- **prefix** uses some particularities of the course of the interpreter with the Prolog language to avoid superfluous calculations.

Both techniques have similar performances (the average computation time is now below 9 seconds) and the required memory space is now 1.5 megabyte for the prefix version. Although the caching version still takes 4 megabytes in the worst case, this one is very simple to use and is the most promising for bigger programs and future use.

The original algorithm was also modified in order to improve the quality of the analysis. First, the *widening* was refined, next, a reexecution strategy was implemented. Both techniques allow a sensible gain of accuracy with reasonnable computation times.

A simpler domain was also used. All the results were discussed previously. On this domain, prefix optimization is still interesting although the caching

performances are impaired because of the simplicity of the abstract operations and the need of time required by caching to manage its internal structures.

Bibliography

[1] S. Abramski and C. Hankin. *Abstract Interpretation of Declarative Languages.* Ellis Horwood Limited, West Sussex, England, 1987.

[2] Ait-kaci. *Warren's Abstract Machine, A tutorial reconstruction.* The MIT Press, Cambridge, Massachusetts, 1991.

[3] P. Boizumault. *Prolog, l'Implantation.* Etudes et Recherches en Informatique. Masson, Paris, 1988.

[4] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 1990. (To Appear).

[5] M. Bruynooghe, G. Janssens, A. Marien, and A. Mulkers. The impact of abstract interpretation: an experiment in code generation.

[6] W.F. Clocksin and C.S. Mellish. *Programming in Prolog.* Springer Verlag, New York, 1981.

[7] P Cousot and R. Cousot. Abstract Interpretation: A unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Conf. Record of Fourth ACM Symposium on POPL*, pages 238–252, Los Angeles, CA, 1977.

[8] K. De Bosschere and L. Wulteputte. Prolog implementation methods. Technical report, Laboratorium voor Elektronica en Meettechniek, 1192.

[9] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Logic Programming. *Journal of Logic Programming*, 8(1-2):75–93, 1990.

[10] Ecole d'été de l'A.F.C.E.T. *La Fiabilité des Programmes*, 1978. Chapter 2 parag. 3.

[11] M. Hermenegildo, R. Warren, and S. Debray. Global Flow Analysis as a Practical Compilation Tool. *Journal of Logic Programming*, 1991. To appear in the Journal of Logic Programming (also published as Technical Report Computer Science Dept, Universidad Politecnica de Madrid, Spain, 1991).

[12] B. Le Charlier. L'analyse statique des programmes par interprétation abstraite. (To appear), 1992.

[13] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. Efficient and Accurate Algorithms for the Abstract Interpretation of Prolog Programs. Research Paper RP-90/9, F.U.N.D.P., University of Namur, August 1990.

[14] B. Le Charlier, K. Musumbu, and P. Van Hentenryck. A Generic Abstract Interpretation Algorithm and its Complexity Analysis (Extended Abstract). In *Eighth International Conference on Logic Programming (ICLP-91)*, Paris (France), June 1991.

[15] B. Le Charlier and P. Van Hentenryck. Experimental Evaluation of a Generic Abstract Interpretation Algorithm for Prolog. In *Fourth IEEE International Conference on Computer Languages (ICCL'92)*, San Fransisco, CA, April 1992.

[16] B. Le Charlier and P. Van Hentenryck. Reexecution in abstract interpretation of prolog. (To appear), 1992.

[17] B. Le Charlier and P. Van Hentenryck. A universal top-down fixpoint algorithm. (To appear), 1992.

[18] C. Livercy. *Theorie des programmes*, pages 18–23. Dunod, 1978.

[19] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, New York, 1984.

[20] K. Musumbu. *Interpretation Abstraite de Programmes Prolog.* PhD thesis, University of Namur (Belgium), September 1990.

[21] P.J. Plauger. *The Standard C Library.* Prentice Hall, 1990.

[22] L. Sterling and E. Shapiro. *The Art of Prolog: Advanced Programming Techniques.* MIT Press, Cambridge, Ma, 1986.

[23] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* Logic Programming Series, The MIT Press, Cambridge, MA, 1989.

[24] P. Van Roy and A.M. Despain. High-performance logic programming with the aquarius porlog compiler. Computer journal.

[25] R. Warren, M. Hermedegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Proc. of the Fifth International Conference on Logic Programming*, pages 684–699, Seattle, WA, August 1988.

[26] V. Englebert, B. Le Charlier, D. Roland and P. Van Hentenryck. Generic Abstract Interpretation Algorithms for Prolog: Two Optimization Techniques and Their Experimental Evaluation. Research Paper CS-91-67, Department of Computer Science, Brown University, December 1991.

Appendix A

Results on the individual operations

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	6	6	0.00	4	1	4.00	83.33
Kalah	325	268	17.54	168	95	1.77	70.77
Queens	35	31	11.43	21	8	2.62	77.14
Press1	2716	2245	17.34	1384	676	2.05	75.11
Press2	869	710	18.30	483	221	2.19	74.57
Peep	457	399	12.69	218	61	3.57	86.65
CS	188	172	8.51	108	43	2.51	77.13
Disj	191	141	26.18	84	43	1.95	77.49
PG	105	97	7.62	62	39	1.59	62.86
Read	1983	1561	21.28	725	425	1.71	78.57
Gabriel	257	219	14.79	149	75	1.99	70.82
Plan	95	89	6.32	51	28	1.82	70.53
QSort	67	56	16.42	36	21	1.71	68.66

Table A.1: Number of Operations on COMPARE

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	8	8	0.00	8	5	1.60	37.50
Kalah	858	678	20.98	522	241	2.17	71.91
Queens	68	52	23.53	52	26	2.00	61.76
Press1	8549	6775	20.75	4767	2953	1.61	65.46
Press2	1953	1533	21.51	1134	601	1.89	69.23
Peep	1396	1284	8.02	693	418	1.66	70.06
CS	468	408	12.82	354	175	2.02	62.61
Disj	472	308	34.75	246	76	3.16	83.90
PG	211	193	8.53	169	95	1.78	54.98
Read	9975	8429	15.50	4783	3510	1.36	64.81
Gabriel	523	414	20.84	371	200	1.85	61.76
Plan	249	227	8.84	171	92	1.86	63.05
QSort	110	81	26.36	59	36	1.64	67.27

Table A.2: Number of Operations on SMALLER

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	1	1	0.00	1	1	1.00	0.00
Kalah	170	118	30.59	170	71	2.39	58.24
Queens	11	7	36.36	11	7	1.57	36.36
Press1	1065	711	33.24	1065	348	3.06	67.32
Press2	353	232	34.28	353	129	2.74	63.46
Peep	227	193	14.98	227	57	3.98	74.89
CS	77	61	20.78	77	45	1.71	41.56
Disj	105	55	47.62	105	34	3.09	67.62
PG	35	29	17.14	35	22	1.59	37.14
Read	1199	840	29.94	1199	191	6.28	84.07
Gabriel	101	66	34.65	101	49	2.06	51.49
Plan	49	43	12.24	49	26	1.88	46.94
QSort	23	14	39.13	23	7	3.39	69.57

Table A.3: Number of Operations on EXTEND

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	0	0		0			
Kalah	81	47	41.98	81	41	1.98	49.38
Queens	12	6	50.00	12	6	2.00	50.00
Press1	265	149	43.77	265	99	2.68	62.64
Press2	85	47	44.71	85	34	2.50	60.00
Peep	39	21	46.15	39	16	2.44	58.97
CS	35	15	57.14	35	13	2.69	62.86
Disj	6	4	33.33	6	3	2.00	50.00
PG	7	3	57.14	7	1	7.00	85.14
Read	383	262	31.59	383	241	1.59	37.08
Gabriel	0	0		0			
Plan	15	9	40.00	15	6	2.50	60.00
QSort	6	2	66.67	6	2	3.00	66.67

Table A.4: Number of Operations on AI_TEST

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	0	0		0			
Kalah	63	42	33.33	63	33	1.91	47.62
Queens	4	2	50.00	4	2	2.00	50.00
Press1	220	134	39.09	220	102	2.16	53.64
Press2	79	43	45.57	79	35	2.26	55.70
Peep	0	0		0			
CS	33	17	48.48	33	16	2.06	51.52
Disj	16	6	62.50	16	5	3.20	68.75
PG	20	12	40.00	20	9	2.22	55.00
Read	22	16	27.27	22	12	1.83	45.45
Gabriel	56	31	44.64	56	26	2.15	53.57
Plan	0	0		0			
QSort	0	0		0			

Table A.5: Number of Operations on AI_IS

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	3	1	66.67	3	1	3.00	66.67
Kalah	22	15	31.82	22	14	1.57	36.36
Queens	0	0		0			
Press1	210	121	42.38	210	102	2.06	51.43
Press2	71	39	45.07	71	37	1.92	47.89
Peep	203	175	13.79	203	167	1.22	17.73
CS	12	6	50.00	12	6	2.00	50.00
Disj	16	8	50.00	16	8	2.00	50.00
PG	13	7	46.15	13	6	2.17	53.85
Read	264	195	26.14	264	191	1.38	27.65
Gabriel	55	32	41.82	55	24	2.29	56.36
Plan	4	2	50.00	4	2	2.00	50.00
QSort	23	14	39.13	23	8	2.88	65.22

Table A.6: Number of Operations on AI_VAR

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	9	3	66.67	9	3	3.00	66.67
Kalah	605	376	37.85	605	335	1.81	44.63
Queens	60	26	56.67	60	25	2.40	58.33
Press1	4726	3042	35.63	4726	2534	1.87	46.38
Press2	1748	1154	33.98	1748	1016	1.72	41.88
Peep	1531	1092	28.67	1531	1082	1.41	29.33
CS	528	252	52.27	528	248	2.13	53.03
Disj	494	202	59.11	494	202	2.45	59.11
PG	186	106	43.01	186	93	2.00	50.00
Read	3463	2503	27.72	3463	2361	1.47	31.82
Gabriel	411	235	42.82	411	222	1.85	45.99
Plan	98	68	30.61	98	66	1.48	32.65
QSort	57	27	52.63	57	21	2.71	63.13

Table A.7: Number of Operations on AI_FUNC

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	3	3	0.00	3	3	1.00	0.00
Kalah	226	174	23.01	226	151	1.50	33.19
Queens	22	18	18.18	22	17	1.29	22.73
Press1	1669	1297	22.29	1669	1037	1.61	37.87
Press2	577	449	22.18	577	389	1.48	32.58
Peep	367	315	14.17	367	291	1.26	20.71
CS	130	114	12.31	130	106	1.23	18.46
Disj	149	99	33.59	149	98	1.52	34.23
PG	62	56	9.68	62	47	1.32	24.19
Read	1495	1122	24.95	1495	990	1.51	33.78
Gabriel	173	138	20.23	173	120	1.44	30.64
Plan	67	61	8.96	67	56	1.20	16.42
QSort	42	33	21.43	42	29	1.45	30.95

Table A.8: Number of Operations on EXTG

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	3	1	66.67	3	1	3.00	66.67
Kalah	226	128	43.36	226	115	1.97	49.12
Queens	22	10	54.55	22	9	2.44	59.09
Press1	1669	962	42.36	1669	716	2.33	57.10
Press2	577	322	44.19	577	263	2.19	52.69
Peep	367	222	39.51	367	212	1.73	42.23
CS	130	76	41.54	130	69	1.88	46.92
Disj	149	69	53.69	149	68	2.19	54.36
PG	62	38	38.71	62	29	2.14	53.23
Read	1495	945	36.79	1495	820	1.82	45.15
Gabriel	173	98	43.35	173	80	2.16	53.76
Plan	67	49	26.87	67	44	1.45	34.33
QSort	42	22	47.62	42	16	2.63	61.90

Table A.9: Number of Operations on RESTRG

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	6	2	66.67	6	2	3.00	66.67
Kalah	229	136	40.61	229	136	1.68	40.61
Queens	29	13	55.17	29	13	2.23	55.17
Press1	1835	1177	35.86	1835	1177	1.56	35.86
Press2	701	458	34.66	701	458	1.53	34.66
Peep	530	380	28.30	530	380	1.39	28.30
CS	153	81	47.06	153	81	1.89	47.06
Disj	124	64	48.39	124	64	1.94	48.39
PG	80	44	45.00	80	44	1.82	45.00
Read	1181	850	28.03	1181	850	1.39	28.03
Gabriel	190	113	40.53	190	113	1.68	40.53
Plan	78	56	28.21	78	56	1.39	28.21
QSort	34	14	58.82	34	14	2.43	58.82

Table A.10: Number of Operations on EXTC

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	6	4	33.33	6	4	1.50	33.33
Kalah	229	182	20.52	229	156	1.47	31.88
Queens	29	21	27.59	29	19	1.53	34.48
Press1	1835	1512	17.60	1835	761	2.41	58.53
Press2	701	585	16.55	701	388	1.81	44.65
Peep	530	473	24.39	530	411	1.29	22.45
CS	153	119	22.22	153	99	1.55	35.29
Disj	124	94	24.19	124	91	1.36	26.61
PG	80	62	22.50	80	47	1.70	41.25
Read	1181	1027	13.04	1181	559	2.11	52.67
Gabriel	190	153	19.47	190	106	1.79	44.21
Plan	78	68	12.82	78	46	1.70	41.03
QSort	34	25	26.47	34	17	2.00	50.00

Table A.11: Number of Operations on RESTRC

	Original	Prefix		Caching			
Program	calls	calls	OR-PR	calls	eval	ratio	OR-eval
Append	11	9	18.18	11	6	1.83	45.45
Kalah	485	455	6.19	485	259	1.87	46.60
Queens	58	50	13.79	58	29	2.00	50.00
Press1	3266	3045	6.77	3266	2000	1.63	38.76
Press2	1209	1140	5.71	1209	674	1.79	44.25
Peep	711	676	4.92	711	522	1.36	26.58
CS	324	290	10.49	324	157	2.06	51.54
Disj	257	227	11.67	257	93	2.76	63.81
PG	166	148	10.84	166	80	2.08	51.81
Read	1993	1899	4.72	1993	1240	1.61	37.78
Gabriel	360	322	10.56	360	176	2.05	51.11
Plan	160	150	6.25	160	75	2.13	53.13
QSort	68	57	16.18	68	38	1.79	44.12

Table A.12: Number of Operations on UNION